Desenvolvimento de Jogos para Android

Explore sua imaginação com o framework Cocos2D

Casa do Código

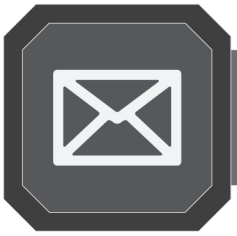
ANDERSON LEITE

© Casa do Código
Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.
Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos mecânicos, gravação ou quaisquer outros.

Casa do Código
Livros para o programador
Rua Vergueiro, 3185 - 8º andar
04101-300 – Vila Mariana – São Paulo – SP – Brasil

Casa do Código
Livros para o programador

Uma editora de livros técnicos feita por desenvolvedores para desenvolvedores.

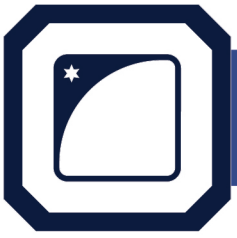


Inscreva-se em nossa newsletter e receba novidades e lançamentos

www.casadocodigo.com.br/newsletter

Curta nossa fanpage no Facebook

www.facebook.com/casadocodigo



Caelum:
Cursos de TI presenciais e online

www.caelum.com.br

Dê seu feedback sobre o livro. Escreva para contato@casadocodigo.com.br

Google Android
Crie aplicações para celulares e tablets
Casa do Código
Aplicações Java para web com JSF e JPA
Casa do Código
Test-Driven Development
Teste e Design no Mundo Real
Casa do Código
Ruby on Rails
Coloque sua aplicação web nos trilhos
Casa do Código
Guia da Startup
Como startups e empresas estabelecidas podem criar produtos web rentáveis
Casa do Código
Web Design Responsivo
Páginas adaptáveis para todos os dispositivos
Casa do Código
iOS: Programe para iPhone e iPad
Casa do Código
RAFAEL STEIL

Já conhece os nossos títulos?

E muito mais em:
www.casadocodigo.com.br

Casa do Código
Livros para o programador

Agradecimentos

Agradeço em especial ao amigo Mauricio Tollin, da BivisSoft, com quem tive a oportunidade de aprender os conceitos iniciais em desenvolvimento de jogos.

Existem muitas outras pessoas que gostaria de agradecer, por me ajudarem de forma direta ou indireta nesse projeto: Adriano Almeida, Alberto Souza, Chris Leite, Delson Leite, Edson Sueyoshi, Elenira Ferreira, Guilherme Silveira, Karine Hermes, Lucia Hermes, Mauricio Aniche, Paulo Silveira, Sheila Paixao e Victor Hermes.

Obrigado a todos vocês.

Sumário

1 Introdução ao desenvolvimento de jogos no Android **1**

1.1 O que você encontrará neste livro 3

1.2 Que comece a diversão! 7

2 Protótipo de um jogo **9**

2.1 Iniciando o projeto 11

2.2 Criando a base do jogo 20

2.3 Desenhando o objeto principal 25

2.4 Captando os comandos do usuário e movendo objetos 29

2.5 Criando o inimigo 33

2.6 Detectando colisões e mostrando resultados 35

2.7 Adicionando um placar 41

2.8 Criando botões de interface do usuário 43

2.9 Adicionando mais vida: imagens da nave e do céu 46

2.10 Conclusão 48

3 História do jogo **51**

3.1 14-bis 52

3.2 14-bis VS 100 Meteoros 53

4 Tela inicial: Lidando com Background, logo e botões de menu **57**

4.1 Iniciando o projeto 59

4.2 Sobre o Cocos2D 63

4.3 Background 64

4.4 Assets da Tela de abertura 66

4.5 Capturando configurações iniciais do device 67
4.6 Logo 70
4.7 Botões 70
4.8 Conclusão 76

5 Tela do jogo e objetos inimigos 79
5.1 GameScene 80
5.2 Transição de telas 82
5.3 Engines 82
5.4 Meteor 85
5.5 Tela do game 86
5.6 Conclusão 88

6 Criando o Player 89
6.1 Desenhando o Player 90
6.2 Botões de controle 92
6.3 Atirando 97
6.4 Movendo o player 102
6.5 Conclusão 104

7 Detectando colisões, pontuando e criando efeitos 107
7.1 Detectando colisões 108
7.2 Efeitos 111
7.3 Player morre 116
7.4 Placar 117
7.5 Conclusão 119

8 Adicionando sons e música 121
8.1 Executando sons 122
8.2 Cache de sons 123
8.3 Música de fundo 124
8.4 Conclusão 125

9 Voando com a gravidade! 127
9.1 Usando o Acelerômetro . . . 128
9.2 Controlando a instabilidade . . . 136
9.3 Calibrando a partir da posição inicialdo aparelho . . . 137
9.4 Desafios com o acelerômetro . . . 139
9.5 Conclusão . . . 139

10 Tela final e game over 141
10.1 Tela final . . . 142
10.2 Tela Game Over . . . 145
10.3 Conclusão . . . 148

11 Pausando o jogo 149
11.1 Montando a tela de pause . . . 150
11.2 Controlando o Game Loop . . . 152
11.3 Adicionando o botão de pause . . . 154
11.4 A interface entre jogo e pause . . . 156
11.5 Pausando o jogo . . . 157
11.6 Pausando os objetos . . . 161
11.7 Conclusão . . . 163

12 Continuando nosso jogo 165
12.1 Utilizando ferramentas sociais . . . 165
12.2 Highscore . . . 166
12.3 Badges . . . 167
12.4 Desafios para você melhorar o jogo . . . 168
12.5 Como ganhar dinheiro? . . . 169
12.6 Conclusão . . . 170

Versão: 16.2.5

Capítulo 1

Introdução ao desenvolvimento de jogos no Android

River Raid, para Atari, foi provavelmente o primeiro jogo de videogame que joguei. Nesse clássico game da `Activision` criado em 1982, o jogador controlava uma nave que se movia de baixo para cima na tela, ganhando pontos por matar inimigos, destruir helicópteros, naves e balões. E mais: era possível encher o tanque passando por estações de gás.

Figura 1.1: RIVER RAID no Atari

Incrível como um desenho simples e 2D podia ser tão divertido. Controlar a nave, fazer pontos e passar por obstáculos me garantiam horas de diversão.

Com o passar do tempo, novos jogos foram surgindo e se tornaram cada vez mais sofisticados. Apesar de todos os conceitos dos jogos antigos terem sido mantidos, um jogo de Playstation 3, por exemplo, pode envolver dezenas de desenvolvedores.

Atualmente, com o crescimento dos *casual gamers*, os celulares e tablets se tornaram plataformas de sucessos e disputadas. Com eles, o desenvolvimento de um jogo não precisa mais de uma quantidade enorme de desenvolvedores. Uma ideia interessante e bem implementada pode ser o suficiente para seu jogo obter sucesso. Só depende de você.

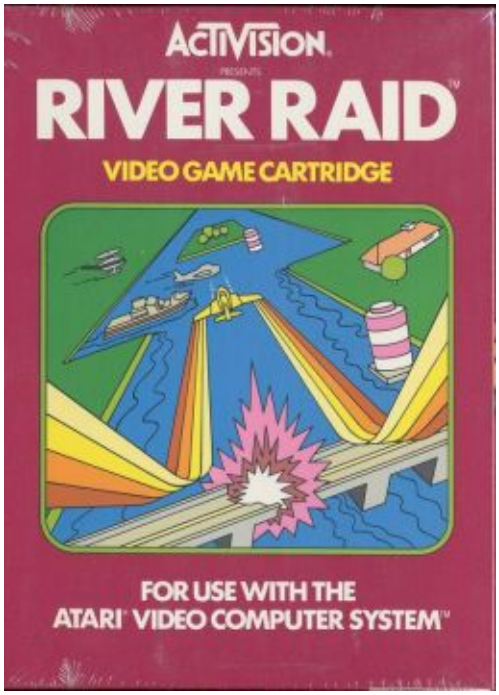


Figura 1.2: Capa do jogo RIVER RAID de 1982

1.1 O que você encontrará neste livro

Este livro é escrito para desenvolvedores que já conhecem a linguagem Java e o básico da plataforma Android. Ele é dividido em 3 partes principais:

- Um protótipo inicial
- Um estudo do jogo que será desenvolvido
- Um jogo desenvolvido com Cocos2D

A ideia é que seja um bom guia para todos aqueles que querem iniciar no desenvolvimento de games, seja profissionalmente, para evoluir seus conhecimentos ou mesmo por pura diversão.

Um protótipo inicial

No início do livro, será desenvolvido um jogo simples, programado com apenas 2 classes. O objetivo é se familiarizar e ter uma noção geral dos conceitos básicos no desenvolvimento de games. Esses conceitos aparecem em quase todos os jogos, sejam eles simples ou avançados.

Nesse capítulo não será utilizado nenhum framework de desenvolvimento, apenas Android puro. Mesmo assim, chegaremos a um resultado bem interessante, como esse:

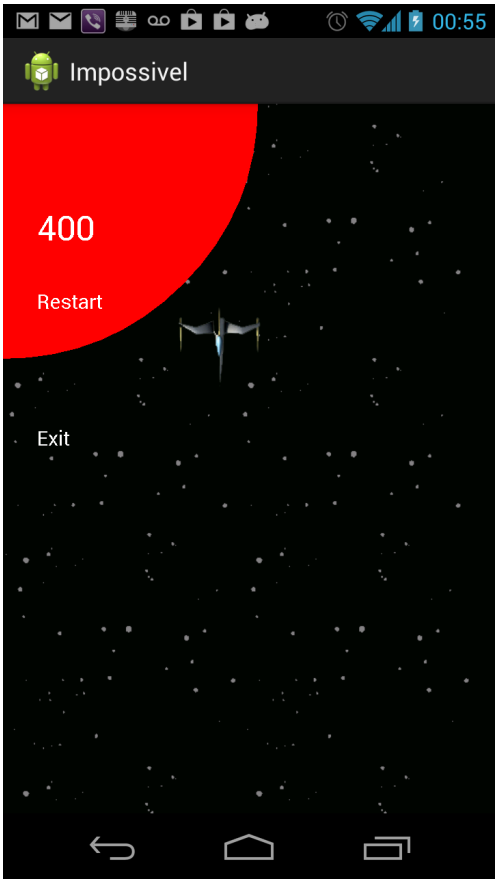


Figura 1.3: Imagem do nosso protótipo.

O código do nosso protótipo pode ser encontrado aqui:
https://github.com/andersonleite/jogos_android_prototipo

Um estudo do jogo que será desenvolvido

Programação é apenas uma parte do desenvolvimento de games. Empresas focadas em desenvolvimento de jogos possuem roteiristas para criar a história dos games, designers para definir o melhor visual do jogo, profissionais de som para a trilha sonora e efeitos, designers de interface para definir como será a experiência do jogador no game, entre outros. O marketing e divulgação são casos à parte.

Teremos um capítulo especial para planejar um pouco a história do jogo, determinar as transições de tela e estudar o visual do jogo a ser desenvolvido, que será nessa direção:

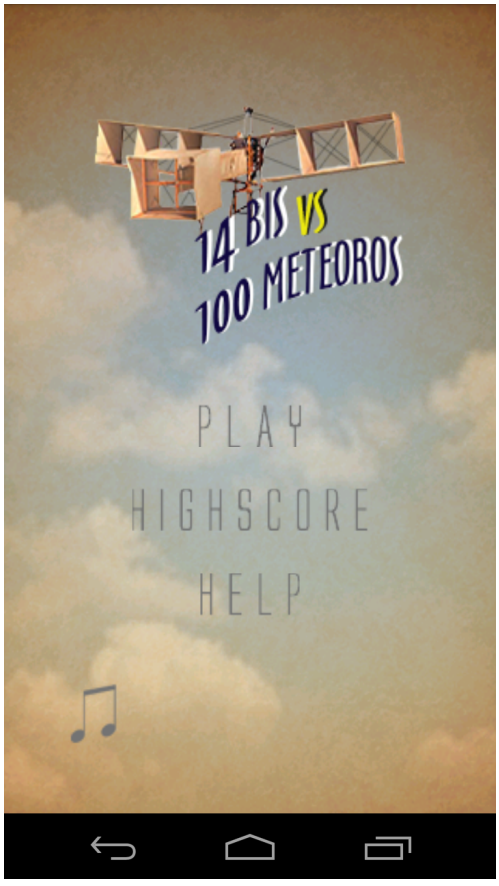


Figura 1.4: 14 bis VS 100 Meteoros

Também veremos um pouco sobre como deixar o jogo viciante e poder ganhar dinheiro com itens, missões e upgrades.

Um jogo desenvolvido com Cocos2D

Quando os principais conceitos já tiverem sido passados e a história e planejamento do jogo finalizada, iniciaremos o desenvolvimento do nosso jogo principal. Para ele, utilizaremos um framework chamado `Cocos2D`, que facilita e otimiza diversas questões usuais no desenvolvimento de jogos.

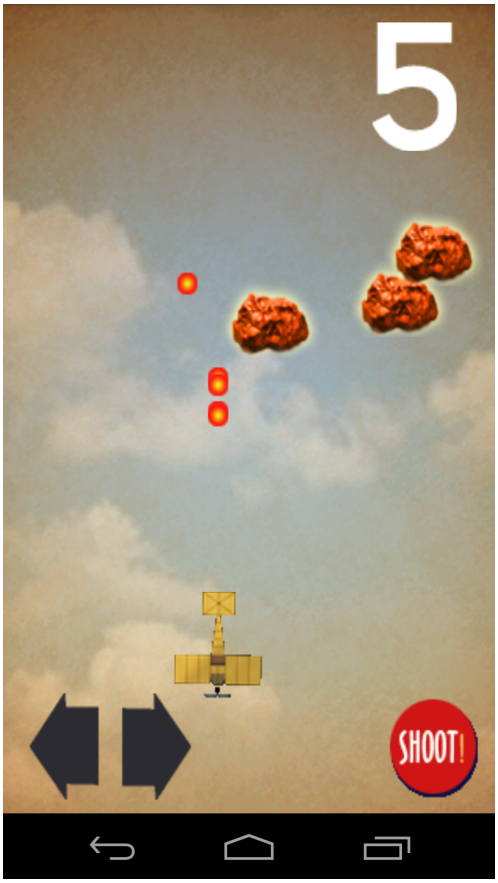


Figura 1.5: 14 bis VS 100 Meteoros

O código do jogo com Cocos2D completo está aqui:
https://github.com/andersonleite/jogos_android_14bis

E o jogo pode ser instalado no seu Android pela Google Play:
https://play.google.com/store/apps/details?id=br.com.casadocodigo.bis

Grupo de Discussão

Existe um grupo de discussão focado exclusivamente para os exemplos que serão desenvolvidos aqui. Caso você tenha dúvidas em algum passo, ou mesmo venha a implementar modificações e criar o seu próprio jogo com o que aprendeu, compartilhe!

https://groups.google.com/group/desenvolvimento-de-jogos-para-android
Você também pode utilizar o forum do GUJ para resolver dúvidas:
http://www.guj.com.br/

1.2 Que comece a diversão!

Este livro vai te dar a base para criar um jogo! Você saberá por onde começar e terá os principais conceitos e a forma de pensar necessária para desenvolver um game 2D ao final dessa leitura. A partir disso, é a sua própria criatividade e determinação que poderão fazer das suas ideias o novo jogo de sucesso no mundo dos games!

Capítulo 2

Protótipo de um jogo

Vamos começar a desenvolver um jogo! Este será um capítulo fundamental para todo o livro, focado em conceitos importantes, ilustrando com muita prática. Nele percorreremos as principais etapas que precisamos ter em mente ao desenvolver um jogo.

Com os conceitos desse capítulo poderemos desenvolver jogos bem interessantes, porém, o objetivo agora é explorarmos as mecânicas por trás dos games e sermos apresentados à forma de pensar necessária.

Para percorrer esse caminho, inciaremos criando um protótipo. Criar um protótipo será bom pelas seguintes razões:

- Conseguiremos um rápido entendimento da visão geral necessária para desenvolver um game.
- Não precisaremos nos preocupar com criar diversas telas que um jogo pode ter, permitindo focar apenas nos conceitos importantes.

- Permitirá entrar em detalhes mais complexos quando de fato iniciarmos nosso game.

Nosso protótipo terá as funcionalidades básicas encontradas nos games, vamos conhecer os objetivos.

Funcionalidades do protótipo

Pense em um jogo 2D tradicional como Super Mario Bros ou mesmo Street Fighter. Eles possuem uma série de semelhanças. Em ambos você controla algum elemento, que podemos chamar de Player. O player recebe algum tipo de estímulo (*input*) para executar movimentos na tela, como teclado, joystick ou mouse. Após os inputs o player pode ganhar pontos se algo acontecer, normalmente associado a encostar em outro objeto do jogo, o que faz com que algum placar seja atualizado. Em determinado momento o player pode ganhar ou perder o jogo, por diversos motivos, como superar um tempo, ultrapassar uma marca de pontos ou encostar em algum outro objeto do game.

Essas são as mecânicas básicas de qualquer jogo. Pense em outro jogo com as características semelhantes e tente fazer esse paralelo. No protótipo que criaremos nesse capítulo, implementaremos essas mecânicas , entendendo como desenvolvê-las em um aplicativo Android.

Nosso jogo terá as seguintes funcionalidades:

- Um player que será representado por uma circunferência verde, posteriormente, a nave.
- Mover o player de acordo com um estímulo, no caso, o toque na tela (input).
- Um inimigo que será representado por uma circunferência que aumentará com o passar do tempo.
- Um placar que será atualizado de acordo com o tempo no qual o player não é capturado pelo inimigo.
- Game Over quando o inimigo encostar no player
- Opções de restart e exit

Ao fim desse capítulo, teremos o protótipo abaixo.

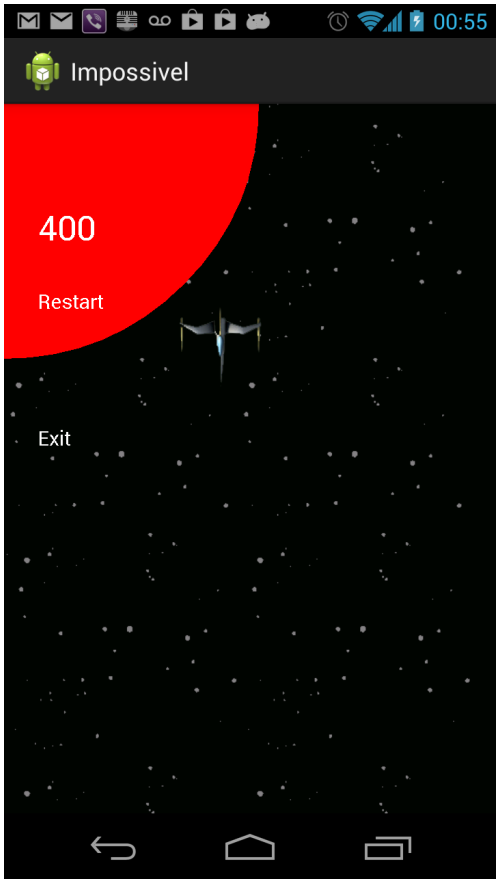


Figura 2.1: Imagem do jogo.

Temos muito para percorrer nesse protótipo. Repare que ao entender a lógica por traz de um jogo, poderemos criar qualquer tipo de game. Vamos ao protótipo!

2.1 Iniciando o projeto

Vamos iniciar criando um projeto comum de Android. Como você já criou algum aplicativo Android, perceberá que o procedimento é o mesmo. Não é necessário configurar nada específico para jogos ao criar o projeto. Lembre-se que você precisa ter o Eclipse e o SDK do Android instalados, que podem ser baixados respectivamente em:

http://www.eclipse.org/ http://developer.android.com/sdk/

Pela página de desenvolvedores do Android, há a possibilidade de baixar um bundle que já traz o Eclipse junto ao plugin, num único download.

Esse é um livro focado em quem já conhece o básico do desenvolvimento Android, mas mesmo assim passaremos passo a passo em alguns pontos e revisaremos conceitos chave, para facilitar seu acompanhamento.

No Eclipse vá em `File` acesse as opções em `New` e selecione `Android Application Project`. Criaremos um projeto chamado *Impossible*. Por que esse nome? Se você reparou na lista de funcionalidades, nosso protótipo nunca tem um final feliz!

Coloque o nome do pacote como `br.com.casadocodigo.impossible`:

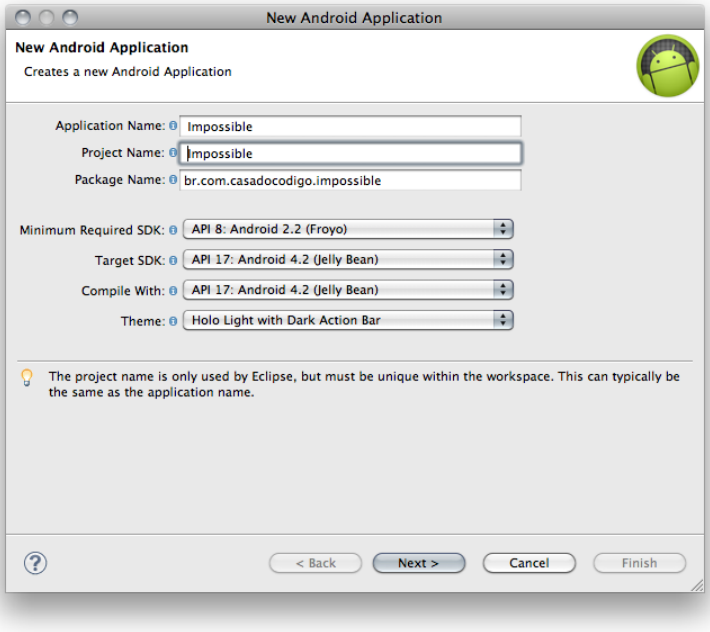

New Android Application
New Android Application
Creates a new Android Application
Application Name: Impossible
Project Name: Impossible
Package Name: br.com.casadocodigo.impossible
Minimum Required SDK: API 8: Android 2.2 (Froyo)
Target SDK: API 17: Android 4.2 (Jelly Bean)
Compile With: API 17: Android 4.2 (Jelly Bean)
Theme: Holo Light with Dark Action Bar
The project name is only used by Eclipse, but must be unique within the workspace. This can typically be the same as the application name.
< Back
Next >
Cancel
Finish

Figura 2.2: Criando o projeto.

No passo 2, deixe selecionadas as opções padrão.

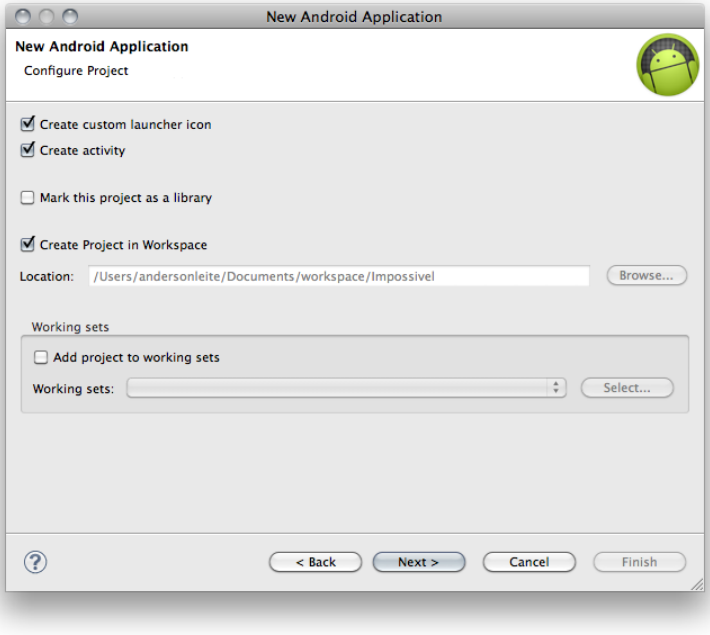


Figura 2.3: Configurações opcionais.

Da mesma forma, mantenha as opções padrão na terceira tela.

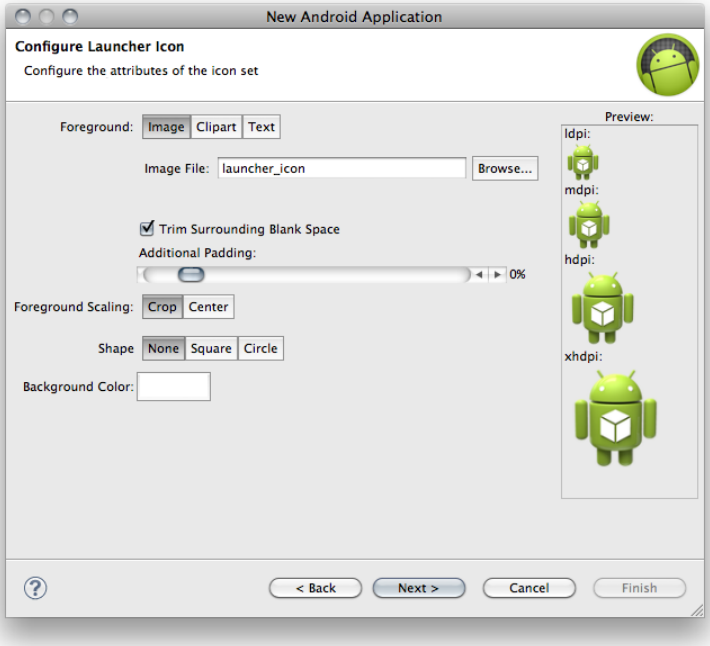


Figura 2.4: Configurações opcionais.

Na quarta tela, selecione a opção `BlankActivity`.

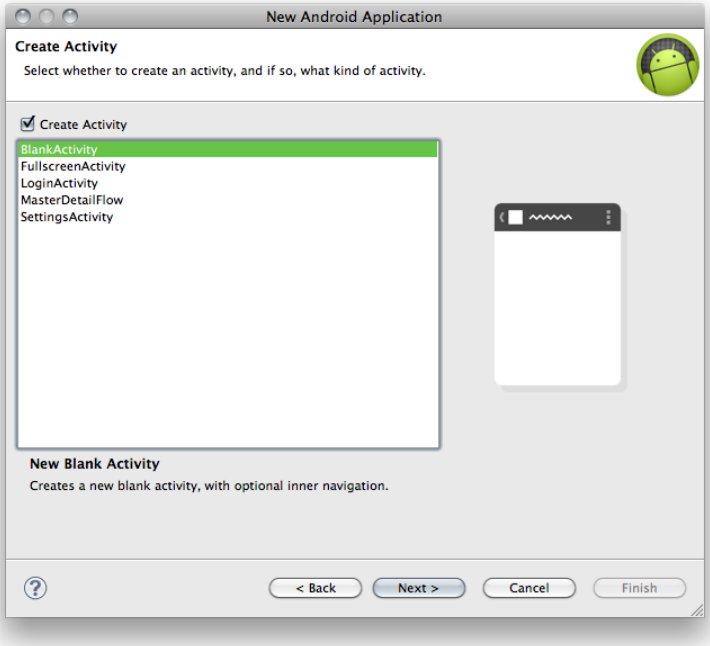


Figura 2.5: Configurações opcionais.

Na última tela crie a Activity com name `Game` e clique em *Finish*.

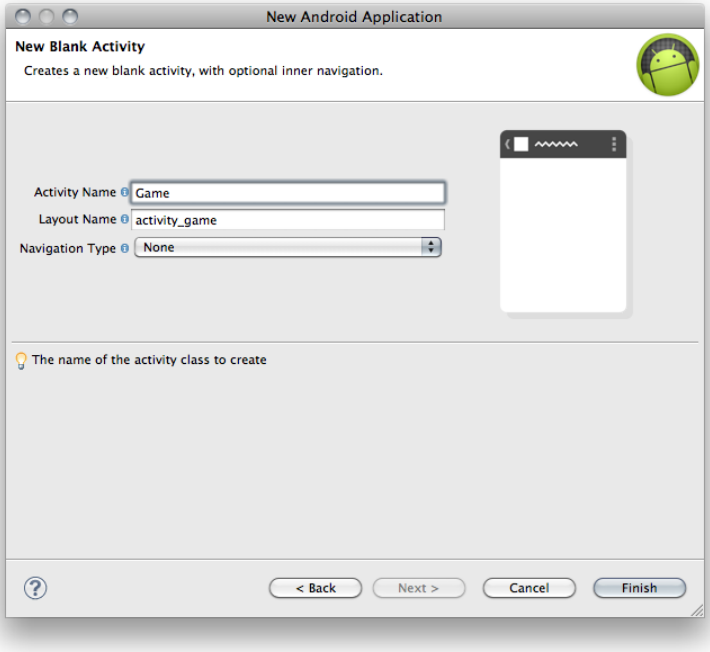


Figura 2.6: Configurações opcionais.

Opcional: Configurando o Emulador

Você tem a opção de rodar os projetos do livro em um aparelho com Android ou utilizando o emulador do Eclipse. É fortemente recomendado que você utilize um aparelho, pois utilizar o emulador pode ser bem lento, além de não dispor de funcionalidades como o acelerômetro que utilizaremos mais para frente.

Para configurar um novo emulador, no Eclipse selecione o menu `Window` e depois `Android Virtual Device Manager`.

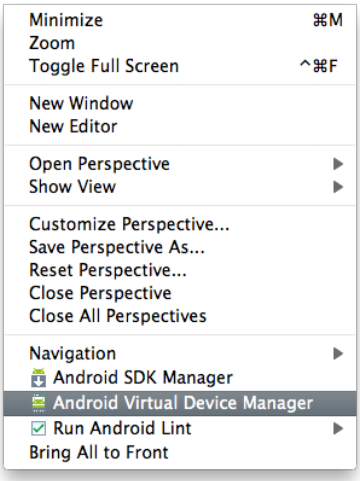


Figura 2.7: Configurando o emulador.

Você deve ver a tela a seguir:

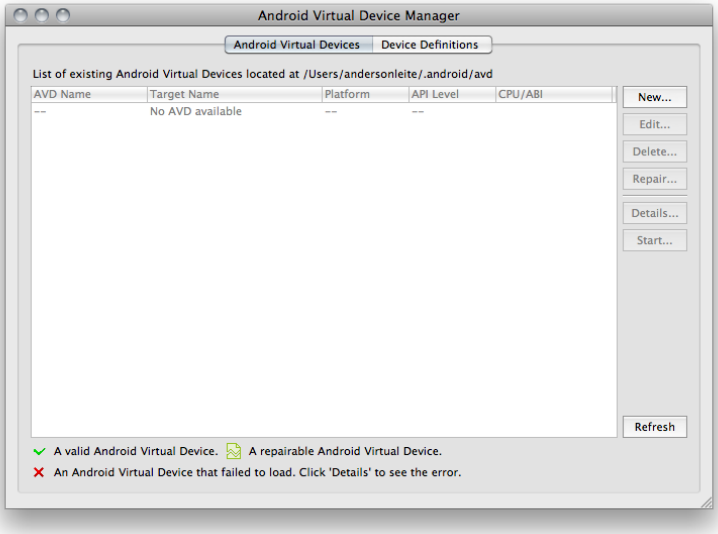


Figura 2.8: Configurando o emulador.

Clique em `New` e configure o emulador com as características abaixo:

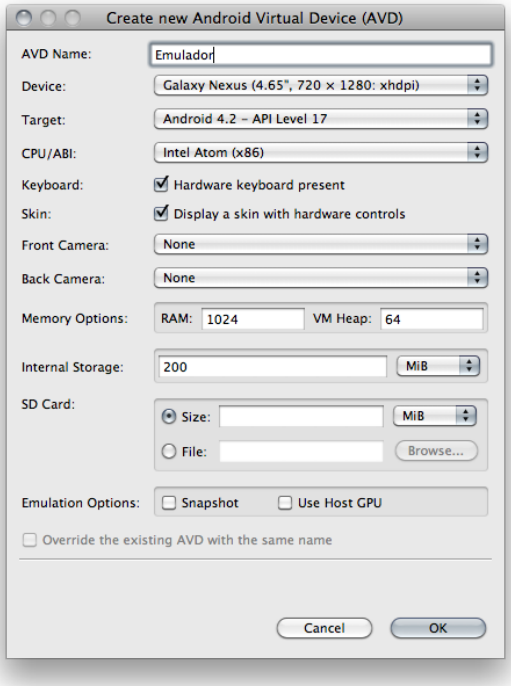


Figura 2.9: Configurando o emulador.

Ao rodar o projeto utilizando o emulador, clicando com o botão da direita no projeto, *Run As* e escolhendo *Android Application*. Você terá uma tela como essa:

Figura 2.10: Configurando o emulador.

Verificando o projeto

Nesse momento temos o projeto iniciado. Você já pode executá-lo: clique com o botão da direita nele, *Run As* e escolha *Android Application*. Ao executá-lo, a tela do emulador deve apresentar algo assim:

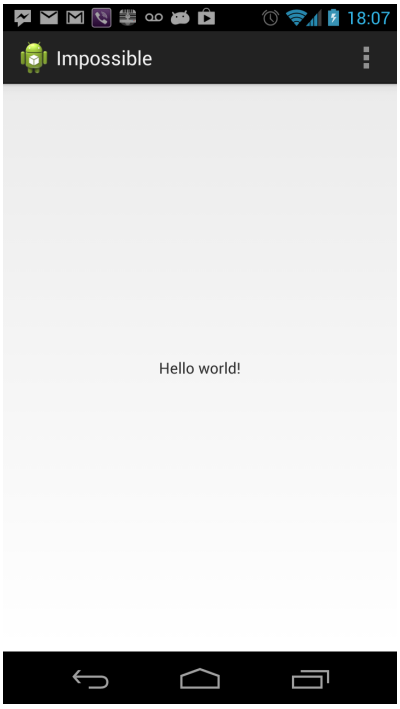


Figura 2.11: Verificando o projeto.

Vamos fazer esse primeiro jogo utilizando posições fixas de X e Y. Para que você obtenha o resultado correto nesse primeiro capítulo, certifique-se de escolher um dispositivo com largura e altura grande de tela. Nos capítulos posteriores aprenderemos a lidar com a chamada fragmentação do Android, possibilitando rodar o jogo em dispositivos de diversos tamanhos de tela.

2.2 Criando a base do jogo

Hora de começar! Para esse protótipo tentaremos manter as coisas simples e diretas, sempre pensando em aplicar os conceitos necessários para criar o jogo nos próximos capítulos.

Teremos apenas duas classes. Uma activity, na qual configuraremos as opções voltadas as configurações do Android e uma classe que terá a lógica do jogo.

É bem comum que a tela inicial de qualquer jogo em Android seja uma Activity, em que teremos as configurações iniciais como determinar os tamanhos de tela, de-

finir o que ocorre quando o usuário da pause ou mesmo que processos devem ser parados quando o usuário sair do jogo.

A lógica de um jogo normalmente é dividida em diversas classes. Aqui a orientação a objeto se faz realmente necessária para um boa utilização dos elementos do game. Nesse primeiro capítulo, manteremos as coisas simples, concentrando a lógica em apenas uma classe. Nos próximos capítulos, quando já tivermos passado pelos conceitos importantes, definiremos a arquitetura do nosso jogo, assim como dividiremos as responsabilidades em diversas classes.

Game Activity

A activity `Game` começará simples. Essa classe é a porta de entrada do nosso jogo, por onde identificamos as características de aplicativos Android, como `inputs` do usuário. Vamos remover o que o Eclipse gerou de código padrão e deixar a classe assim:

```
package br.com.casadocodigo.impossible;

import android.app.Activity;
import android.os.Bundle;

public class Game extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
}
```

Você pode, opcionalmente, usar a tela cheia do aparelho (*Fullscreen*) adicionando as opções abaixo no método `onCreate` da Game Activity:

```
requestWindowFeature(Window.FEATURE_NO_TITLE);
getWindow().setFlags(
        WindowManager.LayoutParams.FLAG_FULLSCREEN,
        WindowManager.LayoutParams.FLAG_FULLSCREEN);
```

A lógica do Impossible: criando um loop infinito

A classe que conterá a lógica se chamará `Impossible`. Essa classe deve herdar de `SurfaceView`, e além disso implementar `Runnable`. Dessa forma poderemos

executar a lógica do jogo em uma thread separada e analisar qualquer estímulo, como por exemplo, os input vindos pelo toque na tela.

```
package br.com.casadocodigo.impossible;

import android.content.Context;
import android.view.SurfaceView;

public class Impossible extends SurfaceView implements Runnable {

    public Impossible(Context context) {
        super(context);
    }

    @Override
    public void run() {
        // TODO Auto-generated method stub

    }
}
```

No `Android` um objeto do tipo `SurfaceView` permite que desenhos sejam executados sobre a superfície em vez de trabalhar com um XML de apresentação.

Agora que já temos a `Activity Game`, que é a porta de entrada, e a classe `Impossible`, que representa a lógica do jogo, vamos criar o link entre elas. Para isso, na activity `Game` iniciaremos uma variável da lógica do jogo, que chamaremos de `view`. Além disso, passaremos essa variável no método `setContentView()` da `Game Activity`, para renderizá-la.

```
package br.com.casadocodigo.impossible;

import android.app.Activity;
import android.os.Bundle;

public class Game extends Activity {
    Impossible view;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

```
        // Lógica do jogo
        view = new Impossible(this);

        // Configura view
        setContentView(view);
    }
}
```

Pode rodar seu projeto novamente, mas ainda não temos bons resultados! Parece que um jogo ainda está longe de se concretizar, mas até o final do capítulo você já terá um protótipo para mostrar.

Game Loop

Normalmente os jogos têm um primeiro conceito importante: `um loop infinito`, conhecido como *game loop* ou *main loop*.

O jogo é uma série de interações nesse loop infinito. Nele, o jogo define posições de elementos, desenha-os na tela, atualiza valores como placar, verifica colisões entre elementos. Isso tudo é realizado diversas vezes por segundo, em que cada tela desenhada é chamada de frame. Uma boa analogia são os desenhos feitos em blocos de papel, onde cada desenho (*frame*) é um pedaço da animação. Ao passar tudo rapidamente temos a impressão de movimento.

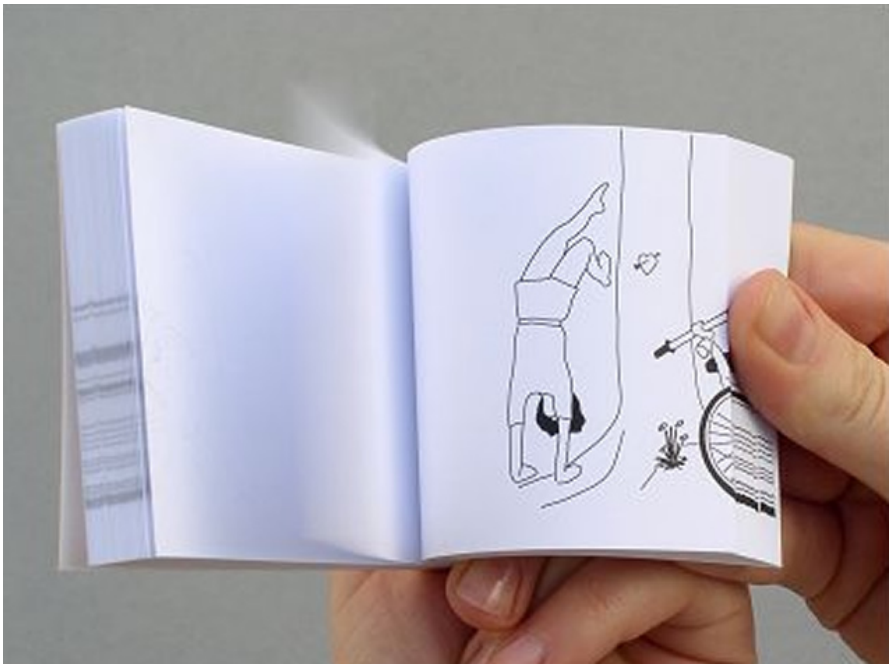


Figura 2.12: Desenho animado em bloco de papel.

Esse conceito é extremamente importante e, com o passar do tempo, difícil de lidar no desenvolvimento de um jogo. Com vários objetos tendo diversas possibilidades na tela a chance de perder o controle é grande. Por isso, tente sempre criar métodos pequenos, com pouca responsabilidade. Dessa forma, encontrar o que pode estar errado fica muito mais fácil.

Vamos colocar o nosso loop infinito dentro do `run` da classe `Impossible`:

```
public class Impossible extends SurfaceView implements Runnable {
    boolean running = false;
    Thread renderThread = null;

    public Impossible(Context context) {
        super(context);
    }

    @Override
    public void run() {
        while(running) {
```

```
                System.out.println("Impossible Running...!");
            }
        }
    }
```

Mas inicialmente esse `boolean running` é falso. Quem vai deixá-lo `true` e criar uma thread para executar o nosso `Runnable`? Vamos criar um trecho de código dentro da própria classe `Impossible`. Já prevendo futuras pausas do jogo, daremos o nome de `resume` a este método. Adicione na classe:

```
public void resume() {
    running = true;
    renderThread = new Thread(this);
    renderThread.start();
}
```

Virou o problema do ovo e da galinha. Quem invoca esse método? Devemos fazer isso logo no início da aplicação.

Após o carregamento básico da activity no método `onCreate`, o Android, por causa do ciclo de vida da activity, invoca o método `onResume`. Vamos invocar o `resume` da nossa `Impossible` nesse método da activity `Game`:

```
protected void onResume() {
    super.onResume();
    view.resume();
}
```

Rode o projeto novamente. Temos nossa primeira saída, ainda não muito empolgante. O console imprime `Impossible Running...!` interminavelmente.

2.3 Desenhando o objeto principal

O motor do nosso jogo já está ligado, funcionando a todo vapor, porém nada acontece na tela. Nosso próximo passo será definir o objeto principal, que chamaremos de Player. Nosso player será bem simples, apenas um elemento gráfico, no caso um círculo. Pode parecer simples mas jogos 2D são objetos triviais que são animados muitas vezes por segundo. No nosso caso temos um círculo, mas podemos trocar por qualquer recurso ou imagem melhor trabalhado para definir um personagem interessante.

Com a popularização dos smartphones e dos jogos casuais, esse tipo de player simples de jogos 2D reapareceu com muita força. O nome atribuído a esses objetos é `Sprite`, que nada mais são que imagens, normalmente retangulares ou mesmo quadradas, com fundos transparentes.

Você pode encontrar `Sprites` de diversos jogos clássicos na internet. Procure no google por 'sprites' mais o nome de um jogo que você goste. Comece a imaginar como esse jogo funciona com o que falamos até aqui.

Figura 2.13: Sprites do famoso Mario Bros.

Utilizando Android Canvas, Paint e a SurfaceView

Para desenhar elementos na tela do jogo no Android, temos algumas opções. Quando desenhamos na vida real, precisamos de ferramentas como pincéis e um lugar para utilizá-las, como papel ou telas. O elemento `Canvas` no Android representa essa tela, na qual podemos desenhar diversas formas ou mesmo Sprites. Para ter acesso a esse elemento, podemos declarar nossa classe como sendo uma tela, por meio da `SurfaceView`.

Ao utilizar uma `SurfaceView`, temos um tipo de `View` especializado em de-

senhar na tela. O principal propósito da `SurfaceView` é fornecer o que precisamos para que uma segunda `Thread` possa renderizar as atualizações do jogo a todo momento. A única restrição ao utilizar essa classe é se certificar de que essa superfície ou tela já está preparada para receber os desenhos. Para essa verificação podemos utilizar o `SurfaceHolder`.

Para desenhar vamos usar a classe `Paint`. Com ela conseguiremos definir elementos como textos, linhas, figuras geométricas, cores e tudo que for referente a colocar os elementos do jogo na tela. Inicializaremos também uma variável do tipo `SurfaceHolder`, que utilizaremos logo em seguida.

Vamos começar configurando nossos primeiros elementos na classe `Impossible`, declarando dois atributos e populando-os no construtor:

```
SurfaceHolder holder;
Paint paint;

public Impossible(Context context) {
    super(context);
    paint = new Paint();
    holder = getHolder();
}
```

E então podemos criar a Thread que terá acesso aos elementos necessários para renderizar as telas do jogo. Repare que aqui verificamos se a superfície já foi preparada, e iniciaremos os desenhos no canvas. Importante perceber que o Canvas deve ser sempre travado e destravado a cada renderização, dessa forma não temos o efeito de *flickering* na animação, com o qual fica visível os passos intermediários dela (apaga tela, redesenha cada elemento, etc).

Além disso, como a `SurfaceView` necessita que a renderização dos desenhos no canvas seja feita por uma outra Thread, o `SurfaceHolder` pode nos ajudar a verificar se a tela já está pronta e preparada para receber os frames do jogo.

```
public void run() {
    while(running) {
        // verifica se a tela já está pronta
        if(!holder.getSurface().isValid())
            continue;

        // bloqueia o canvas
        Canvas canvas = holder.lockCanvas();
```

```
        // desenha o player
        // o que fazer aqui??? já vamos aprender

        // atualiza e libera o canvas
        holder.unlockCanvasAndPost(canvas);
    }
}
```

Finalmente vamos desenhar o player! Utilizaremos nossos pincéis, no caso, a classe `Paint`. Para não deixar o método `run` muito longo, vamos criar um outro método:

```
private void drawPlayer(Canvas canvas) {
    paint.setColor(Color.GREEN);
    canvas.drawCircle(100, 100, 100, paint);
}
```

Agora basta invocar o método `drawPlayer` de dentro do nosso `run`. Repare que só precisamos alterar uma única linha, a que estava com um comentário:

```
public void run() {
    while(running) {
        // verifica se a tela já está pronta
        if(!holder.getSurface().isValid())
            continue;

        // bloqueia o canvas
        Canvas canvas = holder.lockCanvas();

        // desenha o player
        drawPlayer(canvas);

        // atualiza e libera o canvas
        holder.unlockCanvasAndPost(canvas);
    }
}
```

Rode novamente o seu projeto. Obtivemos nosso primeiro resultado!

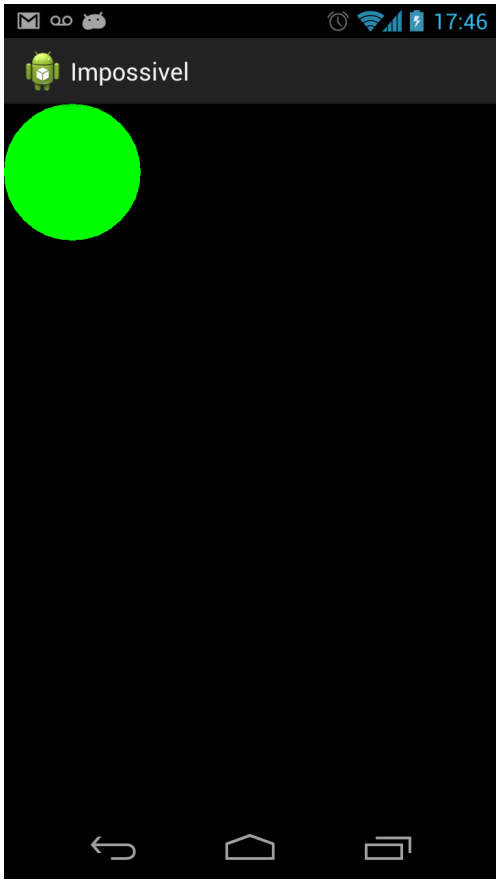


Figura 2.14: Player no Canvas.

2.4 Captando os comandos do usuário e movendo objetos

Existem diversas maneiras de interagir com um jogo e com o player principal. Mover o mouse e clicar, utilizar o teclado, tocar a tela ou mesmo capturar o movimento de um aparelho, usando por exemplo, o acelerômetro. No protótipo, utilizaremos o toque na tela para mover o player. Vamos capturar cada toque como sendo um inputs do usuário, e, a cada vez que isso ocorrer, iremos dar um comando ao nosso jogo.

Nesse momento vamos explorar esse conceito de inputs do usuário no An-

droid, e novamente reparar na importância da Activity principal como porta de entrada do jogo. Utilizaremos uma interface do próprio Android chamada `OnTouchListener`, que tem o método `onTouch` a ser implementado. Toda vez que um toque for detectado, o Android o chama, passando as coordenadas tocadas na superfície da tela. E de posse dessas coordenada, podemos tomar ações sobre os objetos na tela do jogo.

Nesse momento, ao detectar um toque, moveremos para baixo nosso player. Repare que aqui, poderíamos utilizar a informação que recebemos para tomar ações interessantes no jogo, como mover para um lado ou para ou outro, mover mais rápido, etc. Para fim de entendimento de conceito e prototipação, seremos simples nessa implementação.

Antes de mais nada, precisamos saber em que posição nosso player está. Declare o atributo `playerY` no `Impossible`:

```
private int playerY = 300;
```

E, toda vez que invocarem o `drawPlayer`, vamos desenhá-lo nessa altura, em vez daquele número fixo. Altere:

```
private void drawPlayer(Canvas canvas) {
    paint.setColor(Color.GREEN);
    canvas.drawCircle(300, playerY, 50, paint);
}
```

E teremos um método que pode ser invocado para mover o player para baixo (a tela do Android possui a posição `0,0` no canto superior esquerdo):

```
public void moveDown(int pixels) {
    playerY += pixels;
}
```

public class Game extends Activity implements OnTouchListener { ... } [/code]

A classe não vai compilar pois está faltando o método de callback, que é o `onTouch`. Vale lembrar que o Eclipse facilmente escreve a assinatura desse método pra você, clicando no botão de quickfix (é a lâmpada amarela ao lado da linha do erro). Vamos implementá-lo:

```
@Override
public boolean onTouch(View v, MotionEvent event) {
    view.moveDown(10);
```

```
    return true;
}
```

Precisamos avisar o Android de que gostaríamos de receber os inputs de touch. Não basta implementar a interface `OnTouchListener`. Vamos, no `onCreate` da `Game`, avisar ao Android que queremos receber essas notificações da view dentro da própria classe `Game`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // Lógica do jogo
    view = new Impossible(this);

    view.setOnTouchListener(this);

    // Configura view
    setContentView(view);
}
```

Repare que onde o listener vai ficar é uma decisão de design do seu código. Por organização, poderíamos ter criado uma classe específica para lidar com os inputs do dispositivo.

Rode o jogo. Algo já deve ocorrer ao tocar na tela. Nosso player deve ter sua posição alterada. Mas temos um problema, veja como o player se movimenta e o que acontece com a tela:

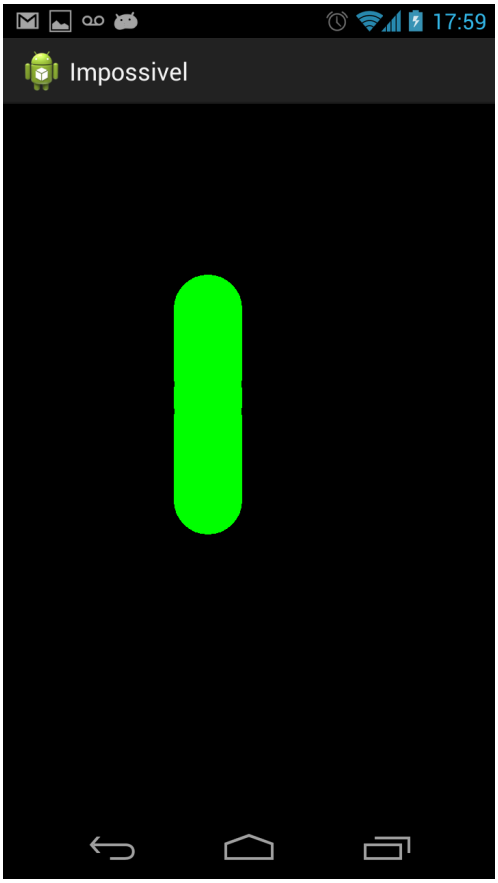


Figura 2.15: Player movendo no canvas, mas temos um problema.

Importante: Limpando a tela

Lembra da relação de um jogo com os desenhos em blocos de papel? O que dá a impressão de movimento em um bloco como esse é que cada imagem é desenhada com uma pequena variação de sua posição anterior. Nenhum desenho, se visto individualmente, da a impressão de movimento ou continuidade do desenho anterior. É como se cada papel do bloco fosse totalmente redesenhado a cada frame.

O que faremos para que o player se mova na tela é zerar a tela toda vez que formos renderizar um novo frame. Como nosso protótipo é simples e não tem um fundo com imagens se movendo, podemos apenas iniciar o frame com um fundo preto. Para jogos com backgrounds mais complexos a estratégia de limpar a tela será

mais complexa.

Na classe `Impossible`, altere o método `run` para pintar a tela de preto, adicionando a linha `canvas.drawColor(Color.BLACK)`:

```
// bloqueia o canvas e prepara tela
Canvas canvas = holder.lockCanvas();
canvas.drawColor(Color.BLACK);
```

Rode agora e toque na tela. A cada toque seu player se moverá!

2.5 Criando o inimigo

Chegamos à parte perigosa! São os inimigos que fazem um jogo ser mais desafiador, que nos instigam a superar algo. O inimigo em um jogo pode estar representado de diversas maneiras. Pode ser o tempo, pode ser uma lógica complexa a ser resolvida ou mesmo outros objetos e personagens.

A partir dos inimigos podemos conhecer diversos conceitos importantes para um jogo funcionar. Assim como o player principal, os inimigos possuem seus próprios movimentos, porém, diferente do player, os movimentos do inimigo costumam ser definidos por lógicas internas do jogo. O interessante é que, por mais que os inputs do usuário não determinem diretamente o movimento do inimigo, quanto mais inteligente ele for de acordo com a movimentação do player, mais interessante e desafiador pode ser o jogo.

Para o protótipo do jogo, nosso inimigo será um outro círculo, porém, como falado acima, esse círculo terá sua própria lógica. Ele crescerá com o tempo, ou seja, de acordo com o passar do jogo, seu raio irá aumentando e consequentemente, ocupando cada vez mais a região do jogo.

Vamos criar, na classe `Impossible`, uma variável que representa o raio do inimigo:

```
private float enemyRadius;
```

Assim como temos um método separado que desenha o player, teremos um que desenha o inimigo. Só que nesse caso especial, queremos que a cada novo frame, o raio do inimigo cresça em 1:

```
private void drawEnemy(Canvas canvas) {
    paint.setColor(Color.GRAY);
    enemyRadius++;
```

```
    canvas.drawCircle(100, 100, enemyRadius, paint);
}
```

Altere o seu `run` e inclua a invocação para desenhar o inimigo:

```
// desenha o player e o inimigo
drawPlayer(canvas);
drawEnemy(canvas);
```

Ao rodar o jogo, nosso inimigo cresce sozinho e o player se afasta com o touch na tela!

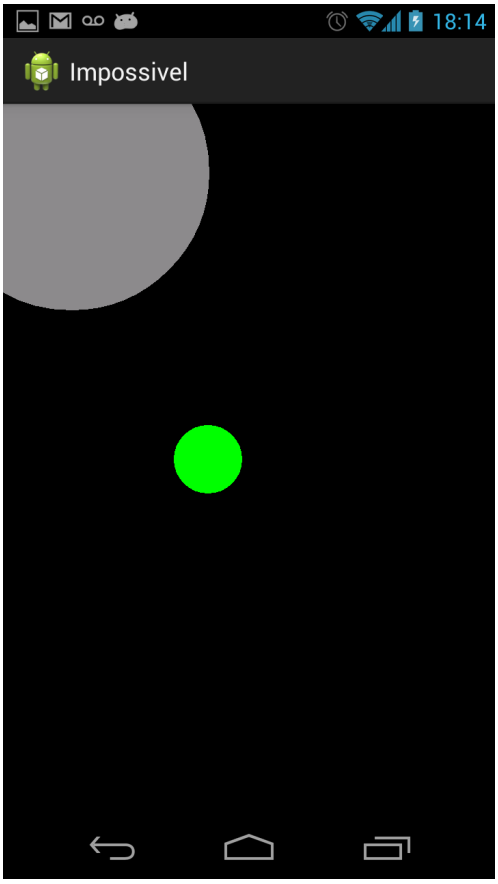


Figura 2.16: Player aparece na tela.

Nesse momento conseguimos mover o player principal e tentar se afastar do ini-

migo que cresce cada vez mais com o passar do tempo. Agora precisamos detectar a colisão!

2.6 Detectando colisões e mostrando resultados

Já passamos pelo conceito de mover objetos, no caso, pelo toque na tela e já também por ter um outro objeto que representa o inimigo e tem sua própria inteligência. A graça do jogo agora é conseguir identificar quando uma determinada situação acontece, situação essa que o player está "lutando contra".

No nosso caso o player não pode encostar no círculo que cresce cada vez mais. Repare que aqui ainda não temos uma história para que essa colisão faça realmente sentido em ser evitada no jogo, porém, é ai que a imaginação faz o jogo se tornar divertido. Jogos antigos, em 2D, não possuíam gráficos incríveis, mas sim, ideias interessantes representadas por objetos simples na tela.

No nosso caso, poderíamos estar desenvolvendo um jogo no qual o player está fugindo de algo. Um exemplo, um vulcão entrou em erupção e nosso herói(player) deve salvar os habitantes dessa vila. Ou seja, sabendo os conceitos, iremos incrementar o visual para que represente uma história interessante.

Detectando colisões

Precisamos então reconhecer que o circulo maior, que representa o inimigo, conseguiu encostar no circulo menor, movido pelo usuário, que representa o player. Detectar colisões é um assunto muito amplo. Existem diversos tipos de detecção de colisões possíveis.

Uma maneira bem tradicional é considerar que cada elemento é um quadrado ou retângulo, verificar através de geometria se um elemento sobrepõe o outro. Essa forma considera mesmo elementos que não contornam um objeto como parte do mesmo. Na imagem abaixo, uma nave de jogos de tiro. Para detectar que algo colide com ela, a área analisada pode ser generalizada para um quadrado ao redor dela.

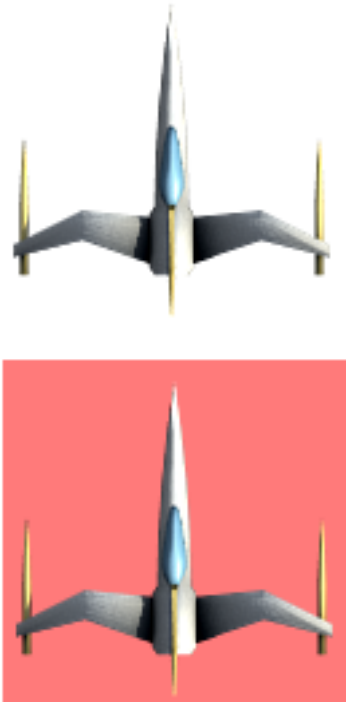


Figura 2.17: Região detectada pelo jogo.

Pode-se questionar se esse método é bom. Será que, se algo encostar na quina do quadrado, que não faz parte da nave, uma colisão será detectada? Em muitos casos essa aproximação é feita por dois motivos.

- Simplificação para detectar a colisão.
- Menor exigência computacional.

Simplificar a detecção por conta de simplificar o algoritmo da colisão é uma prática bem comum, além disso, é bem mais barato computacionalmente do que ter que analisar cada real item de uma imagem de um player.

Colisões no protótipo

Chegamos a um dos conceitos mais importantes no desenvolvimento de um game! Precisamos identificar a colisão entre o player e o inimigo. Esse é o item

chave do nosso protótipo e normalmente na maioria dos jogos. Existem diversas formas de pontuar, e muitas delas utilizam a colisão entre dois ou mais objetos para isso. Jogos de tiro pontuam pela colisão do tiro com o objeto atirado. Jogos como Super Mario Bros e Street Fighter pontuam pelas colisões do player com moedas ou com inimigos.

Existem diversas formas de detectar colisões, algumas mais complexas outras mais simples. Para o nosso protótipo, utilizaremos a colisão de duas circunferências.

A colisão de dois círculos é uma das mais simples, porém, é relacionada a alguns conceitos matemáticos como o `Teorema de Pitágoras`.

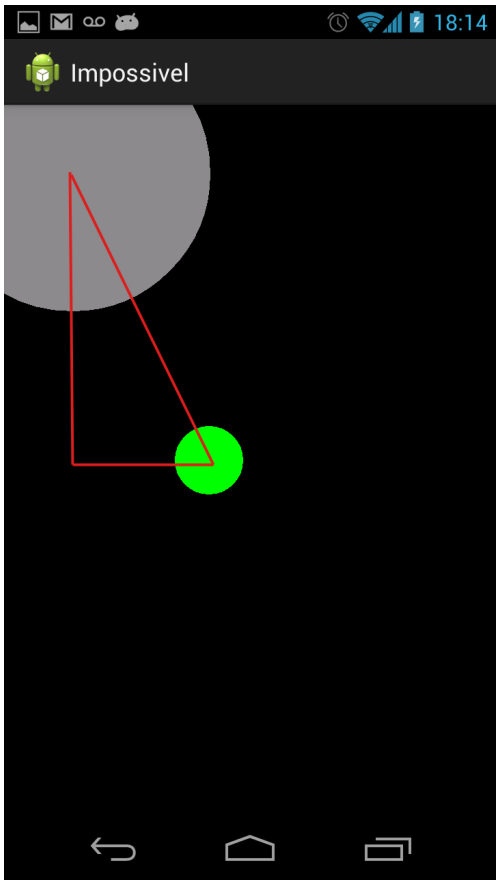


Figura 2.18: Teorema de Pitágoras.

Na figura anterior, existe uma maneira simples matematicamente de determinar

se as circunferências estão sobrepostas. Precisamos identificar os valores a seguir:

- Soma dos raios das duas circunferências
- Valor da hipotenusa, ou distância entre os dois raios

De posse das duas informações acima, conseguimos identificar se a soma dos raios é maior que a hipotenusa gerada. Se for maior, não existe colisão.

Vamos ao código!

Primeiro criaremos algumas variáveis para esse cálculo. As variáveis se referem às posições X e Y de ambas as circunferências, tanto do player quanto a do inimigo. A hipotenusa será chamada de *distance*. Também teremos mais uma variável, *gameover*, caso a colisão seja detectada.

Altere sua classe `Impossible` para ter todos esses atributos. Repare que `enemyRadius` e `playerY` já possuíamos:

```
private int enemyX, enemyY, enemyRadius = 50;
private int playerX = 300, playerY = 300, playerRadius = 50;
private double distance;
private boolean gameover;
```

Refatore os métodos criados anteriormente para que utilizem as variáveis que foram criadas:

```
private void drawEnemy(Canvas canvas) {
    paint.setColor(Color.GRAY);
    enemyRadius++;
    canvas.drawCircle(enemyX, enemyY, enemyRadius, paint);
}

private void drawPlayer(Canvas canvas) {
    paint.setColor(Color.GREEN);
    canvas.drawCircle(playerX, playerY, 50, paint);
}
```

O método que identifica a colisão segue a matemática que já vimos e utiliza a classe `Math` para raiz e potenciação.

```
private void checkCollision(Canvas canvas) {
    // calcula a hipotenusa
```

```
    distance = Math.pow(playerY - enemyY, 2)
            + Math.pow(playerX - enemyX, 2);
    distance = Math.sqrt(distance);

    // verifica distancia entre os raios
    if (distance <= playerRadius + enemyRadius) {
        gameover = true;
    }
}
```

Adicione a chamada ao método que detectará a colisão, dentro do nosso loop do `run`:

```
// desenha o player e o inimigo
drawPlayer(canvas);
drawEnemy(canvas);

// detecta colisão
checkCollision(canvas);
```

Rode o jogo, o que acontece quando houver colisão? Nada! Pois apenas mudamos a variável `gameover` para `true`. Precisamos verificar isso em nosso loop! Devemos confirmar se o jogo terminou, não apenas checar a colisão:

```
// desenha o player e o inimigo
drawPlayer(canvas);
drawEnemy(canvas);

// detecta colisão
checkCollision(canvas);

if(gameover) {
    break;
}
```

E com isso saímos do main loop, congelando a tela e o processamento dos objetos. Teste o jogo!

Seria mais bonito um gameover mais convincente, não? Vamos colocar na tela uma mensagem de Game Over. Crie o método `stopGame` que vai utilizar funções básicas do canvas que ainda não vimos. O objeto `Paint` possui o controle de cor, tamanho e estilo. Utilizaremos o método `drawText` para escrever a mensagem:

```
private void stopGame(Canvas canvas) {
    paint.setStyle(Style.FILL);
    paint.setColor(Color.LTGRAY);
    paint.setTextSize(100);
    canvas.drawText("GAME OVER!", 50, 150, paint);
}
```

Agora altere o `if` do `gameover`:

```
if(gameover) {
    stopGame(canvas);
    break;
}
```

A partir daqui, o jogo já detecta as colisões e para em Game Over caso o inimigo alcance o player. Essa é uma maneira de se pensar em games que foi utilizada por muito tempo. Atualmente, com o avanço do hardware e com velocidades de processamento cada vez maiores, detecções de colisões muito mais complexas foram criadas.

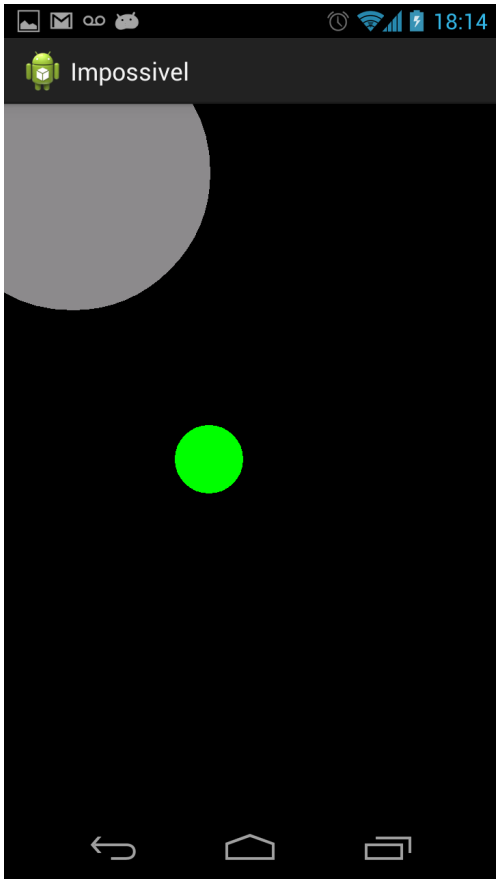


Figura 2.19: Player aparece na tela.

2.7 Adicionando um placar

Vamos ver agora um próximo conceito importante para a maioria dos jogos, a atualização do placar. A maioria dos jogos atualiza alguma forma de pontuação de tempos em tempos. Essa atualização de placar pode ocorrer de diversas formas, por exemplo pela detecção de colisões entre objetos. No protótipo o placar será simples, a cada movimento do player ganharemos pontos.

A ideia é simples, a cada toque na tela o player se afasta do inimigo e, com isso, ganhamos pontos.

Crie o campo `score` na `Impossible`, além de um método que aumenta o

score:

```
private int score;

public void addScore(int points) {
    score += points;
}
```

No método que recebe o evento de toque, incrementaremos os pontos. Atualize o método `onTouch` da classe `Game`:

```
view.moveDown(10);
view.addScore(100);
```

Crie o método `drawScore`. Nele, atualizaremos um campo na tela com o valor atual da variável `score`.

```
private void drawScore(Canvas canvas) {
    paint.setStyle(Style.FILL);
    paint.setColor(Color.WHITE);
    paint.setTextSize(50);
    canvas.drawText(String.valueOf(score), 50, 200, paint);
}
```

No método `run`, após detectar o fim do jogo, faça a invocação ao `drawscore`:

```
// detecta colisão
checkCollision(canvas);

if(gameover) {
    stopGame(canvas);
    break;
}

// atualiza o placar
drawScore(canvas);
```

O jogo deve estar assim:

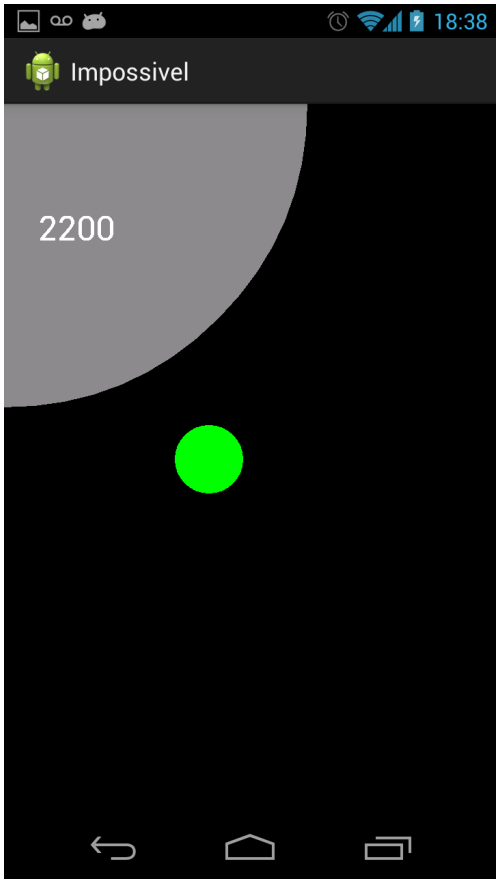


Figura 2.20: Player aparece na tela.

2.8 Criando botões de interface do usuário

Outro conceito importante em jogos é a possibilidade de interagir com elementos de configuração do game. Poder pausar o jogo, entrar em uma tela para mudar, por exemplo, a dificuldade ou mesmo fechar e sair do jogo são partes importante do desenvolvimento.

No protótipo, ilustraremos esse conceito com duas opções na tela, para restart e exit. Simplificaremos para entender o conceito, e depois, poderemos converter para botões mais interessantes.

Na classe `Impossible` crie o método `drawButtons` para desenhar o Restart

e Exit:

```
private void drawButtons(Canvas canvas) {
    // Restart
    paint.setStyle(Style.FILL);
    paint.setColor(Color.WHITE);
    paint.setTextSize(30);
    canvas.drawText("Restart",  50, 300, paint);

    // Exit
    paint.setStyle(Style.FILL);
    paint.setColor(Color.WHITE);
    paint.setTextSize(30);
    canvas.drawText("Exit", 50, 500, paint);
}
```

Com os métodos prontos, faça a chamada para desenhar os botões no método `run` da classe `Impossible`.

```
// atualiza o placar
drawScore(canvas);

// Restart e Exit
drawButtons(canvas);
```

Agora precisaremos de uma forma de reinicializar as propriedades do game. Ainda na classe `Impossible` crie o método que reinicializa as variáveis. Repare que, para o restart, utilizaremos uma abordagem simples de reinicialização.

Crie o método `init` na classe `Impossible`:

```
public void init() {
    enemyX = enemyY = enemyRadius = 0;
    playerX = playerY = 300;
    playerRadius = 50;
    gameover = false;
}
```

Precisamos agora reconhecer o touch em determinada parte da tela e fazer a chamada para o método que criamos.

Para o botão `Exit` utilizaremos o comando `System.exit(0);` que fecha o programa.

Na `Game Activity`, verifique quando os botões recebem o touch no método `onTouch`:

```
@Override
public boolean onTouch(View v, MotionEvent event) {
    if(event.getX() < 100 && event.getY() > 290 &&  event.getY() < 310) {
        view.init();
    }

    // Exit
    if(event.getX() < 100 && event.getY() > 490 &&  event.getY() < 510) {
        System.exit(0);
    }

     // Incrementa em 10 pixels a posição
     // vertical do player e o placar
    view.moveDown(10);
    view.addScore(100);

    return true;
}
```

A tela final do protótipo deve estar assim:

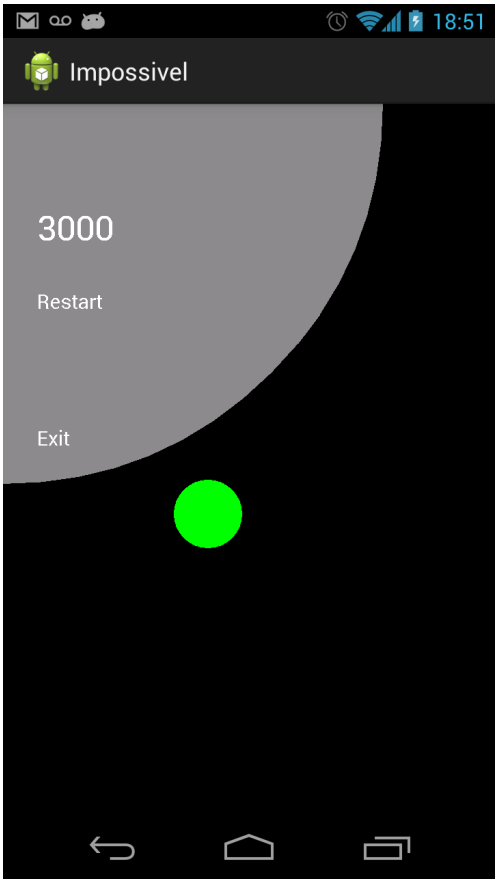


Figura 2.21: Protótipo final.

2.9 Adicionando mais vida: imagens da nave e do céu

Temos toda a lógica do protótipo rodando e já podemos, finalmente, alterar alguns elementos visuais para finalizar o protótipo do jogo e fechar os conceitos básicos.

Vamos desenhar um background que simula um céu escuro com estrelas. Para isso, utilizaremos a imagem `sky.png`. Um ponto importante é reparar que quanto mais sprites forem adicionados ao jogo, mais esforço computacional, o que pode tornar o jogo mais lento. Mais a frente, utilizaremos frameworks que otimizam essa questão.

Adicione os arquivos `sky.png` e `nave.png` no diretório `assets` do projeto.

Vamos alterar algumas linhas para utilizar as imagens. No método `run` da classe `Impossible` adicione uma linha que funcionará como background do game.

```
//    canvas.drawColor(Color.BLACK);
canvas.drawBitmap(BitmapFactory.decodeResource(getResources(),
                    R.drawable.sky), 0, 0, null);
```

Altere o método `drawPlayer` para renderizar a imagem da nave.

```
private void drawPlayer(Canvas canvas) {
    paint.setColor(Color.GREEN);

    canvas.drawBitmap(BitmapFactory.decodeResource(getResources(),
              R.drawable.nave), playerX-50, playerY-50, null);
    // canvas.drawCircle(playerX, playerY, 50, paint);
}
```

Altere a a cor do inimigo para vermelho.

```
private void drawEnemy(Canvas canvas) {
    paint.setColor(Color.RED);
    enemyRadius++;
    canvas.drawCircle(enemyX, enemyY, enemyRadius, paint);
}
```

Pode rodar o jogo, o protótipo está com sprites, e roda os conceitos fundamentais de um jogo!

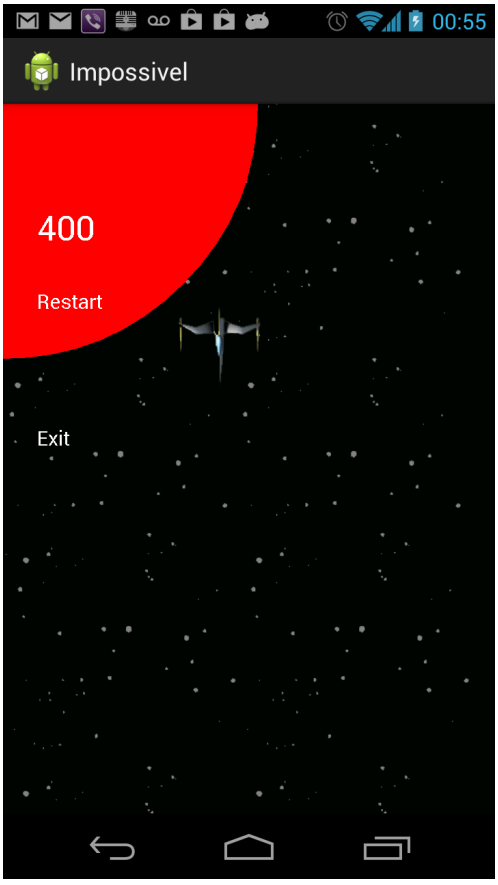


Figura 2.22: Imagem do jogo.

2.10 Conclusão

Um jogo possui diversos conceitos bem específicos, que não são comuns em outros tipos de projetos de software como web ou desktop. Para desenvolver um jogo é interessante ter bem esclarecidas as partes principais que compõem o quebra-cabeça do mundo da programação para games.

Um jogo ocorre normalmente em um `loop infinito`, no qual `inputs`, ou seja, entradas de comandos, são interpretados e utilizados para executar as lógicas do game. O movimento do player, normalmente uma imagem chamada `sprite` costuma ser dado a partir desses inputs, assim como o movimento dos inimigos,

que indiretamente, também é calculado.

Conceitos periféricos como `atualização da tela`, `limpeza da tela` e `botões de comandos` também são importantes e devem ser todos muito bem pensados na execução do jogo.

Com isso em mente, podemos planejar nosso jogo e iniciar seu desenvolvimento.

Capítulo 3

História do jogo

Jogos são feitos de magia, de ilusão, de fantasia. São histórias que envolvem as pessoas de uma forma poderosa, na qual o usuário se sente o protogonista estando no comando das ações.

No começo do livro falamos sobre um jogo fantástico chamado `River Raid`, além de desenvolver um protótipo de jogo de avião no capítulo anterior. Pois bem, chegou a hora!

Mas se criarmos um jogo de nave, qual apelo ele terá? O que o diferenciará dos mil outros jogos que podemos encontrar na Play Store? O que fará prender a atenção do jogador?

O enredo, os personagens e a carisma são peças fundamentais que vestem um jogo. É realmente importante considerar um tema chamativo que seja diferente do que já estamos acostumados. Como fazer algo um pouco diferente em um jogo de naves? Como trazer isso para um contexto com o qual os nossos jogadores estejam familiarizados?

Criaremos um jogo também com a temática de aviões, como uma homenagem

a um importante brasileiro que participou do inicio dessa revolução aérea.

Em 1873 nascia Alberto Santos Dumont, um piloto, atleta e inventor brasileiro.

Figura 3.1: Alberto Santos Dumont

Santos Dumont projetou, construiu e voou os primeiros balões dirigíveis com motor a gasolina, conquistando o Prêmio Deutsch em 1901, quando contornou a Torre Eiffel. Se tornou então uma das pessoas mais famosas do mundo durante o século XX.

3.1 14-BIS

Em 1906, Santos Dumont criou um avião híbrido chamado 14 bis, considerado o primeiro objeto mais pesado que o ar a superar a gravidade terrestre.

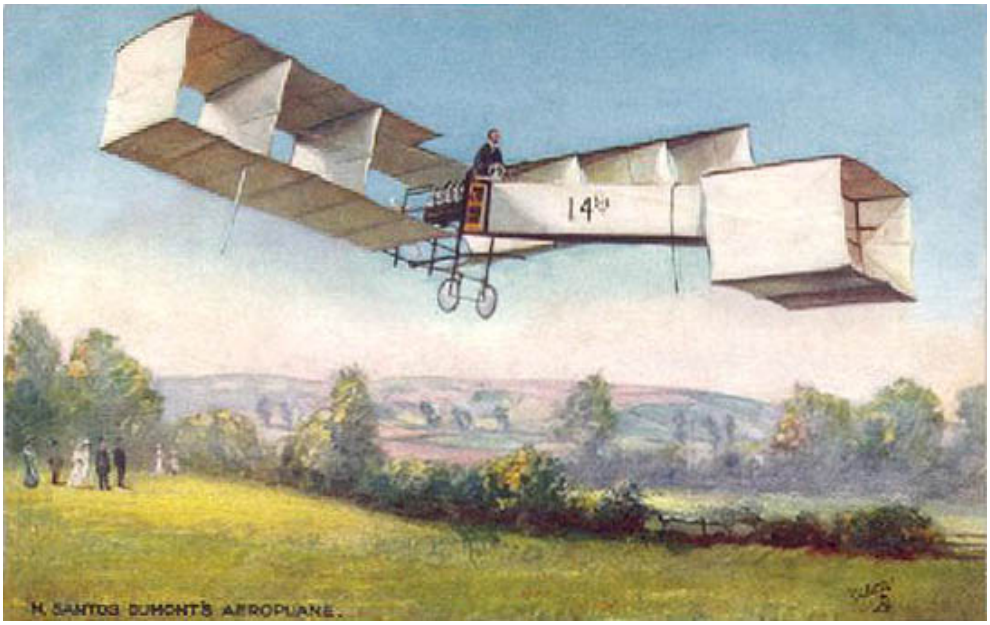


Figura 3.2: 14-bis do brasileiro Santos Dumont

O pesadelo de Santos Dumont

Em agosto de 1914 começava a Primeira Guerra Mundial e os aviões começaram a ser utilizados em combates aéreos. A guerra ficou cada vez mais violenta, com metralhadoras e bombas. Santos Dumont viu seu sonho se transformar em pesadelo.

Em 1932, um conflito entre o estado de São Paulo e o governo de Getúlio Vargas foi iniciado e aviões atacaram a cidade. Essa visão causou muita angustia a Santos Dumont, que se suicidou.

3.2 14-bis VS 100 Meteoros

Santos Dumont inventou o 14-bis não com o intuito de guerra. Nesse jogo, homenagearemos Dumont e sua invenção utilizando sua aeronave para salvar o planeta!

Depois do meteoro Shoemaker-Levy-9 que caiu em Júpiter, depois do meteoro que caiu na Rússia em 2013, tudo indicava que o fim estava próximo. O exército brasileiro detectou a presença de 100 meteoros entrando na órbita terrestre! Esses meteoros acabarão com a existência de toda forma de vida que conhecemos caso não sejam detidos urgentemente.

Figura 3.3: 14-bis VS 100 Meteoros

O planeta está em apuros! Todas as partes do mundo só falam nesse assunto e buscam formas de evitar o fim. Eis que surge um brasileiro, com seu invento 14-bis, para enfrentar o perigo e tentar salvar a todos nós. Munido de uma arma poderosa para destruir os meteoros que caem sobre a terra, você comandará a aeronave 14-bis nessa aventura!

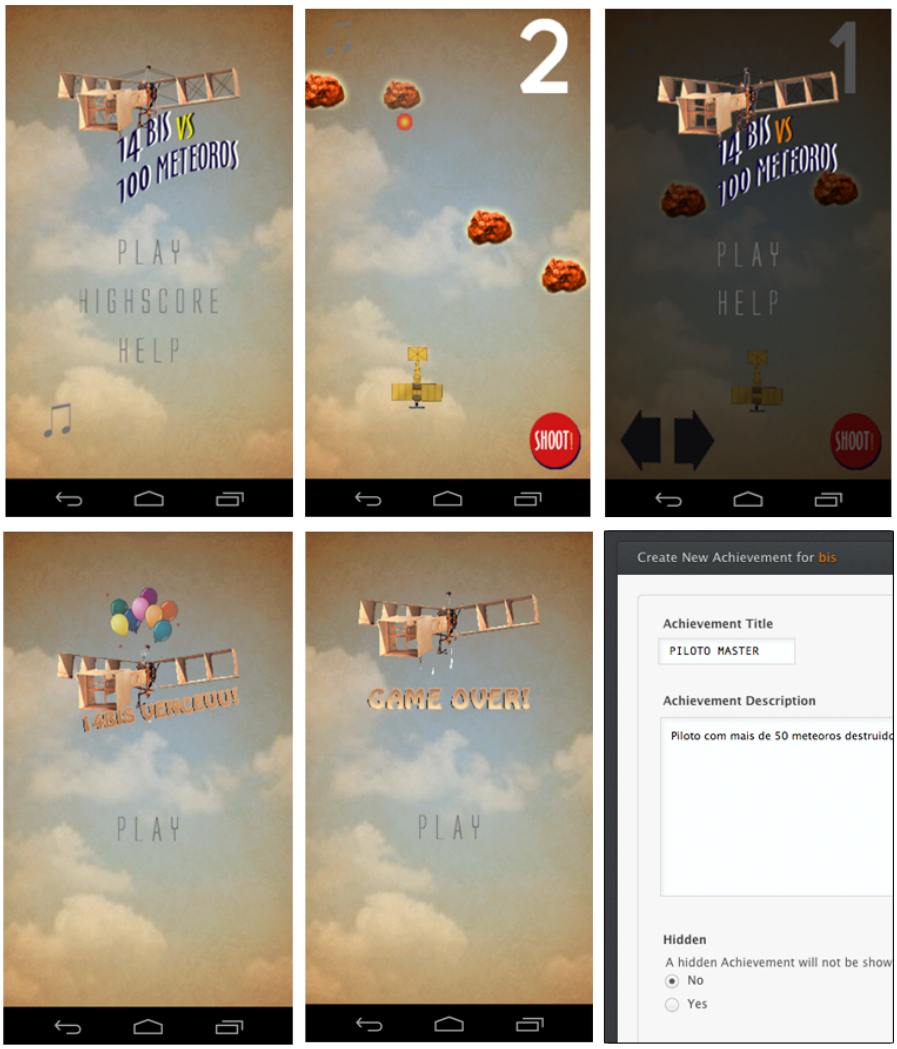


Figura 3.4: Fluxo do game 14-bis VS 100 Meteoros

Capítulo 4

Tela inicial: Lidando com Background, logo e botões de menu

Hora de começar o jogo de verdade! Agora que já passamos pelos conceitos básicos de desenvolvimento de jogos como game loop, sprites, colisões e inputs, podemos ir um pouco mais a fundo no desenvolvimento.

Vamos criar um jogo baseado no game do capítulo anterior, porém dessa vez utilizando um framework de desenvolvimento de jogos chamado Cocos2D. O motivo de utilizar um framework daqui pra frente é otimizar diversos aspectos, entre eles:

- Não se preocupar com a posição exata em pixels dos objetos, como botões, por exemplo
- Utilizar comportamentos já implementados para Sprites, para não ter problemas com posicionamento das imagens
- Eliminar da lógica a questão da limpeza de tela, deixando isso como responsabilidade do framework

- Conseguir efeitos interessantes já implementados pelo Cocos2D
- Trabalhar mais facilmente com sons e efeitos

Nesse capítulo criaremos a tela inicial. Essa tela será composta por um logo, um background, e quatro botões. Veremos aqui como posicionar cada um desses elementos na tela e como detectar os inputs dos botões utilizando o Cocos2D. Ao fim do capítulo, devemos ter a tela inicial como a seguir:

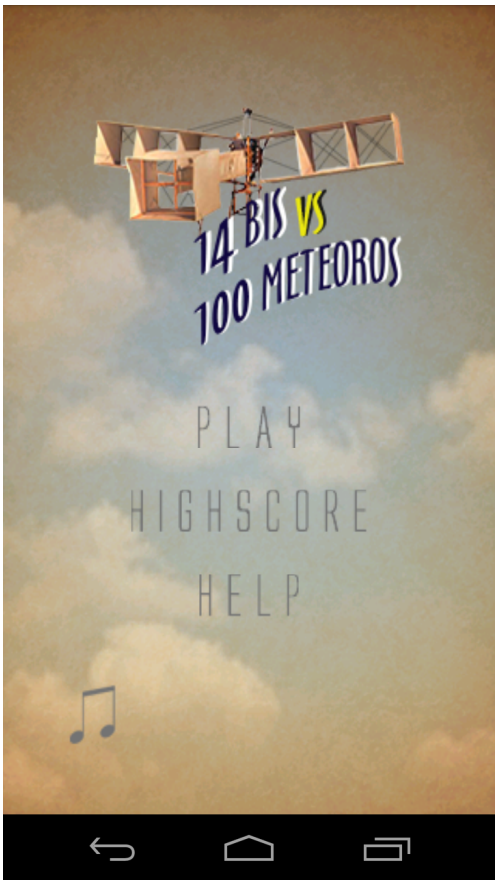


Figura 4.1: Tela de abertura.

Você poderá encontrar o código completo do jogo, junto com algumas melhorias, no meu GitHub:

https://github.com/andersonleite/jogos_android_14bis

Mas prefira você mesmo escrevê-lo! Será menos trabalhoso do que você imagina, e certamente ajudará muito no seu aprendizado.

4.1 Iniciando o projeto

No Eclipse vá em `File` acesse as opções em `New` e selecione `Android Application Project`. Criaremos um projeto chamado *Bis*. Os pacotes ficarão a seu critério, mas você pode seguir a sugestão de usar `br.com.casadocodigo.bis`. Dessa forma você poderá sempre acompanhar com facilidade o código fonte completo que está no github em https://github.com/andersonleite/jogos_android_14bis.

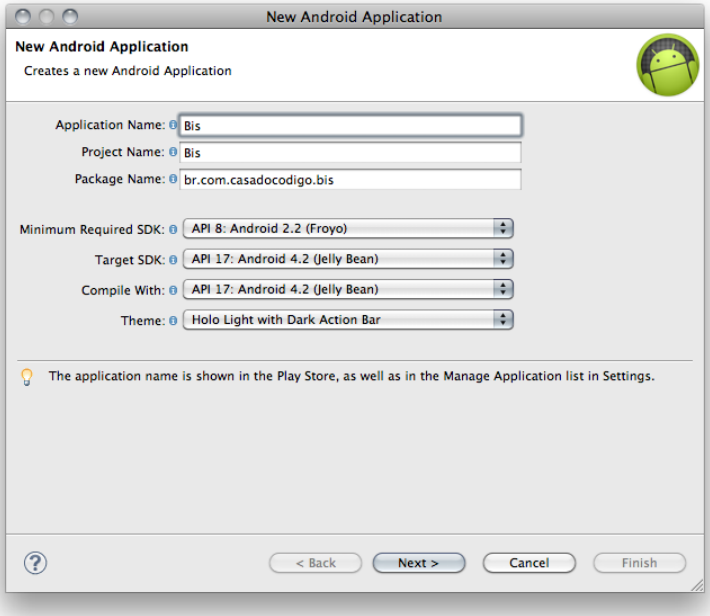

New Android Application
New Android Application
Creates a new Android Application
Application Name: Bis
Project Name: Bis
Package Name: br.com.casadocodigo.bis
Minimum Required SDK: API 8: Android 2.2 (Froyo)
Target SDK: API 17: Android 4.2 (Jelly Bean)
Compile With: API 17: Android 4.2 (Jelly Bean)
Theme: Holo Light with Dark Action Bar
The application name is shown in the Play Store, as well as in the Manage Application list in Settings.
< Back
Next >
Cancel
Finish

Figura 4.2: Criando o jogo 14 bis.

No passo 2, deixe selecionadas as opções padrão.

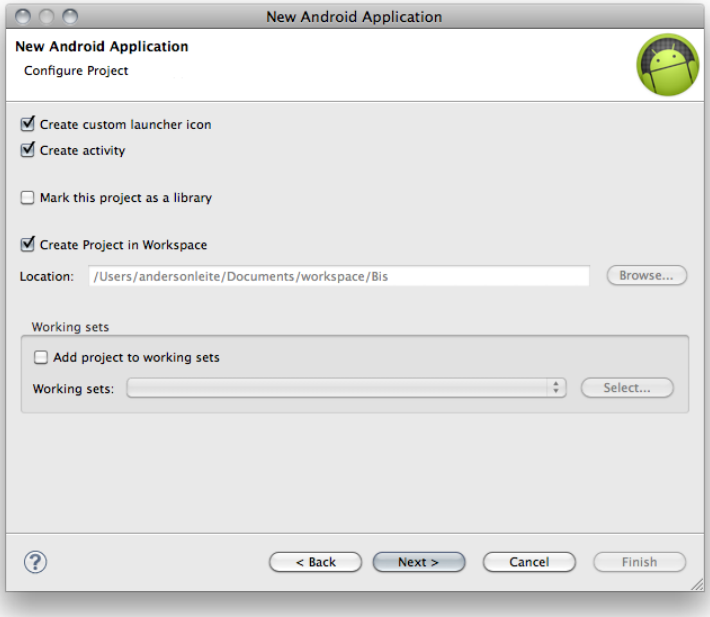


Figura 4.3: Criando o jogo 14 bis.

Na terceira tela você pode configurar o ícone do game. Você encontrará essas imagens em https://github.com/andersonleite/jogos_android_14bis/tree/master/res/

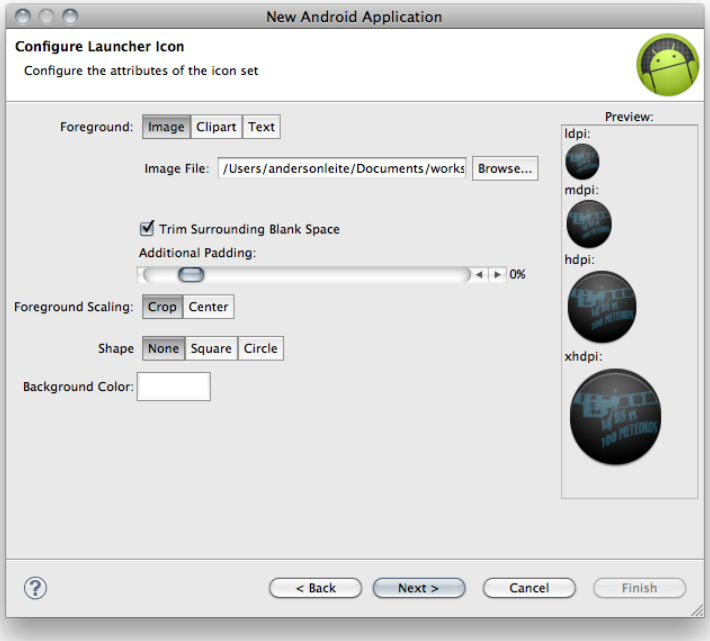


Figura 4.4: Criando o jogo 14 bis.

Na quarta tela, selecione a opção `BlankActivity`.

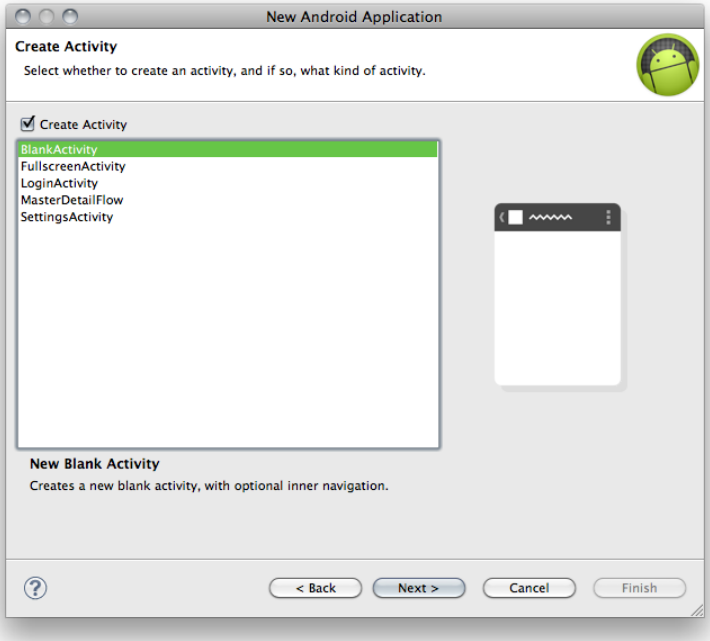


Figura 4.5: Criando o jogo 14 bis.

Na última tela crie a Activity com name `MainActivity` e clique em *Finish*.

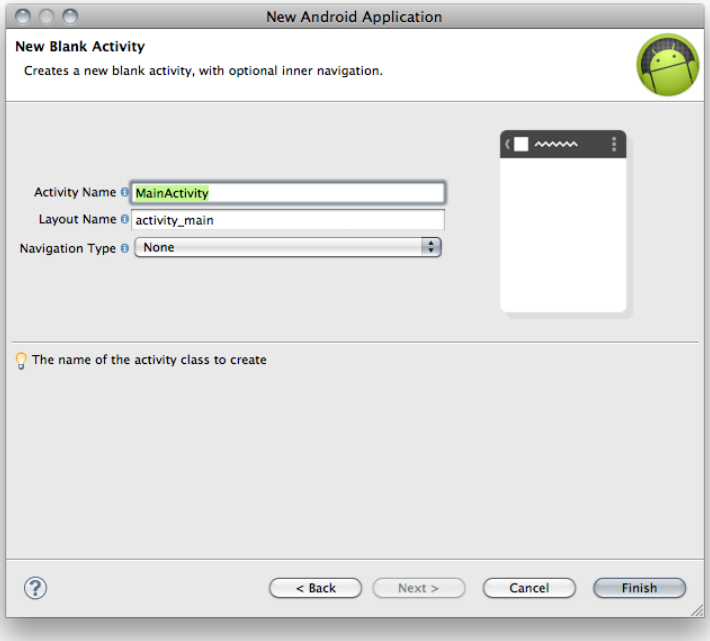


Figura 4.6: Criando o jogo 14 bis.

4.2 Sobre o Cocos2D

O Cocos2D é framework open source de desenvolvimento de jogos. A versão original foi criada em Python e desde então foi portada para diversas linguagens como C++, JavaScript, Objective-C e Java. É muito poderoso e simples. Como se trata de um framework portado de outra língua, sua documentação não está bem organizada e você pode sofrer um pouco para encontrar algumas informações.

Para utilizar a versão Java, basta baixar o arquivo `cocos2D-android.jar` no seguinte repositório:

https://code.google.com/p/cocos2d-android-1/downloads/list

Ou é possível baixar a versão já testada com o jogo no github do livro, em:

https://github.com/andersonleite/jogos_android_14bis/tree/master/libs

Depois, adicione o jar ao `Build Path` do seu projeto. Lembra de como fazer isso no Eclipse? Caso sua view *Package Explorer* esteja aberta, basta dar um clique da direita no jar e depois escolher *Add to Build Path*. Se preferir, dê um clique da direita

no seu projeto, escolha *Build Path* e *Configure Build Path* para escolher e adicionar o Cocos2D.

4.3 Background

A primeira tela do game é a tela de abertura, e no Cocos2D, utilizamos uma classe chamada `CCLayer` para identificar cada tela do jogo. Ao herdar dessa classe do framework, ganhamos alguns reconhecimentos do Coco2D, como conseguir fazer a transição entre as telas com apenas uma linha de código.

Uma classe que herda de `CCLayer` não precisa ter muitos códigos do framework, podemos criar nossa tela inicial como bem entendermos, apenas utilizando esse comportamento para informar ao framework que tipo de objetos estamos representando.

Layers

Criar telas com o `CCLayer` do Cocos2D é criar telas pensando em camadas que se sobrepõem. Essas camadas são transparentes, a menos quando definidas de outra forma, e quando colocadas uma sobre as outras definem a tela final.

Na tela de abertura, podemos pensar em camadas para a imagem de background, para o logo e para o menu.

Criaremos uma classe chamada `TitleScreen`. Você pode criá-la no pacote que desejar. No nosso projeto, utilizamos `br.com.casadocodigo.meteoros.scenes`. Lembre-se de organizar suas classes em pacotes significativos.

Nesta classe, utilizaremos um segundo componente do Cocos2D. Para instanciar uma tela no framework, utilizamos a classe `CCScene`, que é devolvida já pronta para utilizarmos quando invocamos o método `node()`.

Scenes

Outro objeto importante do Cocos2D são as `Scenes`. Com elas, conseguimos inicializar telas do jogo. Um jogo pode ter quantas `Scenes` forem necessárias, porém apenas uma estará ativa por vez.

Por exemplo, no nosso jogo teremos a tela de abertura, a tela do jogo, a tela de ajuda, a tela de pause, etc. Cada uma delas é um `Scene`.

Vamos ao código inicial da tela de abertura. Precisamos de uma classe que saiba trabalhar com camadas, e precisamos de uma tela. Criaremos a classe

`TitleScreen` que receberá essas definições de camadas, e a adicionaremos em uma `Scene`, formando a base da tela inicial.

```
public class TitleScreen extends CCLayer {

    public CCScene scene() {
        CCScene scene = CCScene.node();
        scene.addChild(this);
        return scene;
    }
}
```

O código anterior prepara a tela para utilização e posicionamento dos elementos, no nosso caso, esses elementos serão background, logo, e botões.

Vamos iniciar configurando o background do game. Assim como botões ou logo, o background também é um objeto representado por uma imagem. Lembre-se que para manipular imagens temos o conceito de `Sprites`, que basicamente é um objeto associado a uma figura.

Sprites

Um `Sprite` no Cocos2D é como qualquer outro Sprite, ou seja, uma imagem 2D que pode ser movida, rotacionada, alterar sua escala, animada, etc. Umas das vantagens de utilizar `Sprites` como objetos do Cocos2D é que ganhamos algumas possibilidades de animação, que veremos mais à frente.

Criaremos então a classe `ScreenBackground` e para informar ao Cocos2D que tipo de objeto estamos representando, no caso um `Sprite`, herdaremos de `CCSprite`.

Ganhamos aqui uma facilidade muito importante ao tratar de imagens, que é o posicionamento automático do Cocos2D, com o qual apenas precisamos passar o `asset` desejado.

```
public class ScreenBackground extends CCSprite {

    public ScreenBackground(String image) {
        super(image);
    }
}
```

Uma vez configurado o objeto que representa o background, adicioná-lo à tela de abertura é simples. Instanciamos um objeto do tipo `ScreenBackground` e configuramos sua posição. Aqui, utilizaremos o tamanho da tela, tanto largura quanto altura, para posicionar o background de forma centralizada. Faremos isso com um elemento muito importante do `Cocos2D`, o `CCDirector`, que será aprofundado mais à frente. Basta adicionar o background na tela de abertura, e manteremos um atributo para ele:

```
public class TitleScreen extends CCLayer {

    private ScreenBackground background;

    public TitleScreen() {
        this.background = new ScreenBackground("background.png");
        this.background.setPosition(
            screenResolution(CGPoint.ccp(
                CCDirector.sharedDirector().winSize().width / 2.0f,
                CCDirector.sharedDirector().winSize().height / 2.0f
            )));
        this.addChild(this.background);

    }
    // restante do código
```

Precisamos então do `background.png` do nosso jogo. Criamos algumas imagens e utilizamos outras de fontes gratuitas para nosso jogo. Você deve baixar um zip que as contém nesse endereço:

https://github.com/andersonleite/jogos_android_14bis

Coloque o arquivo `background.jpg` dentro do diretório `assets` do seu projeto. Você precisará repetir esse procedimento para outras imagens, sons e arquivos dos quais precisaremos no decorrer do desenvolvimento de nosso jogo.

4.4 Assets da Tela de abertura

A tela de abertura do game terá 6 assets (arquivos como imagens e figuras) que serão utilizados para compor logo e menus.

Para começar, vamos organizar na classe `Assets` esses arquivos de imagens do game. Assim, podemos fazer chamadas as imagens que estão na pasta de mesmo nome, `Assets` no projeto.

```
public class Assets {
    public static String BACKGROUND = "background.png";
    public static String LOGO = "logo.png";
    public static String PLAY = "play.png";
    public static String HIGHSCORE = "highscore.png";
    public static String HELP = "help.png";
    public static String SOUND = "sound.png";
}
```

Altere a linha da `TitleScreen` que chamava o `background.png` para utilizar a classe `Assets`:

```
this.background = new ScreenBackground(Assets.BACKGROUND);
```

Utilizaremos essa classe para adicionar outros assets mais pra frente, quando os objetos inimigos e o player forem desenhados. É importante ter uma classe como essa para não espalhar suas imagens pelo código. Por exemplo, imagine que você quer alterar a imagem da nave principal. É melhor alterá-la em apenas um lugar, e fazer referência a essa variável nas classes necessárias.

4.5 Capturando configurações iniciais do device

Diversos devices rodam Android atualmente, o que faz com que o tamanho da tela não seja um padrão com largura e alturas fixas. Existem algumas técnicas para tentar limitar esse problema durante o desenvolvimento do jogo. Utilizaremos aqui uma técnica simples para adaptar nosso conteúdo aos diversos dispositivos, capturando as medidas e utilizando-os sempre que for necessário lidar com essa questão.

Para iniciar as configurações de tela e criar a tela inicial, criaremos alguns métodos:

- screenResolution: Recebe a posição do objeto como um tipo do Cocos2D, o GCPoint
- screenWidth: Retorna a largura da tela
- screenHeight: Retorna a altura da tela

O Cocos2D nos ajuda nesse momento, pois já possui objetos preparados para executar essa função. Podemos utilizar a classe `CCDirector` para conseguir os parâmetros da tela.

Director

O CCDirector é um componente que cuida das transições entre `scenes`, ou seja, transições de telas do jogo. Ele é um `Singleton` que sabe qual tela está ativa no momento e gerencia uma pilha de telas, aguardando suas chamadas para fazer as transições.

Vamos implementar a classe `DeviceSettings`, responsável por acessar o `CCDirector` e retornar as medidas e configurações do device.

```
public class DeviceSettings {
    public static CGPoint screenResolution(CGPoint gcPoint) {
        return gcPoint;
    }

    public static float screenWidth() {
        return winSize().width;
    }

    public static float screenHeight() {
        return winSize().height;
    }

    public static CGSize winSize() {
        return CCDirector.sharedDirector().winSize();
    }
}
```

Com isso, podemos refatorar o construtor da classe TitleScreen para ficar como a seguir.

```
import static
    br.com.casadocodigo.nave.config.DeviceSettings.screenHeight;
import static
    br.com.casadocodigo.nave.config.DeviceSettings.screenWidth;

public TitleScreen() {
    this.background = new ScreenBackground(Assets.BACKGROUND);
    this.background.setPosition(
        screenResolution(CGPoint.ccp(
            screenWidth() / 2.0f,
            screenHeight() / 2.0f
```

```
        )));
    this.addChild(this.background);

}
```

O `CCDirector` é também responsável pela inicialização da tela de abertura.

Iniciando a tela de abertura

Tela inicial preparada! Agora precisamos fazer a transição, ou seja, ao chegar na `Activity`, que é a porta de entrada do game, devemos informar ao Cocos2D para iniciar a tela de abertura.

Um objeto muito importante do Cocos2D será utilizado para esse controle. Utilizaremos o `CCDirector` novamente, dessa vez para rodar uma nova tela, a partir do método `runWithScene()` passando como parâmetro a `TitleScreen`, que é a tela de abertura.

```
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // definindo orientação como landscape
        setRequestedOrientation(
            ActivityInfo.SCREEN_ORIENTATION_PORTRAIT);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        getWindow().setFlags(
            WindowManager.LayoutParams.FLAG_FULLSCREEN,
            WindowManager.LayoutParams.FLAG_FULLSCREEN);

        // configura a tela
        CCGLSurfaceView glSurfaceView = new CCGLSurfaceView(this);
        setContentView(glSurfaceView);
        CCDirector.sharedDirector().attachInView(glSurfaceView);

        // configura CCDirector
        CCDirector.sharedDirector().setScreenSize(320, 480);

        // abre tela principal
        CCScene scene = new TitleScreen().scene();
        CCDirector.sharedDirector().runWithScene(scene);
```

```
    }
}
```

Já é possível rodar o projeto e ver a tela de abertura com o background configurado! Faça o teste.

4.6 Logo

Vamos utilizar a mesma ideia e colocar um logo do jogo no topo da tela.

O logo é uma imagem simples. E imagens são coordenadas por objetos que chamamos de `Sprites`. Criaremos um `Sprite` de forma simples para posicionar o logo e utilizaremos o método `setPosition` para que o Cocos2D saiba onde colocar o elemento.

Pra finalizar, basta adicionar o logo a tela inicial pelo método `addChild()`. Mude o construtor da sua `TitleScreen`:

```
public TitleScreen() {

    // código atual
    ...

    CCSprite title = CCSprite.sprite(Assets.LOGO);
    title.setPosition(screenResolution(
        CGPoint.ccp( screenWidth() /2 , screenHeight() - 130 )));
    this.addChild(title);
}
```

Ao rodar o projeto já temos imagem de background e logo do jogo posicionados.

4.7 Botões

Os botões são partes importantíssimas do jogo. É partir deles que o usuário interage com o game e que recebemos comandos para transicioná-las e, mais à frente, mover o player, atirar, etc.

Para colocar os botões na tela inicial, utilizaremos conceitos do Cocos2D e conceitos de Orientação a Objetos, além de Design Patterns, como Decorator e Observable.

Utilizando o Cocos2D o trabalho com inputs de botões fica bem mais simples, não precisando detectar a posição do toque na tela e comparar com o posicionamento dos `Sprites`. Esse trabalho será feito pelo framework, e o jogo pode se preocupar com a lógica em si.

Vamos ao código. Para informar ao framework que o objeto que estamos representando é um botão que pode ser pressionado e precisa responder a eventos, herdaremos da classe `CCLayer`. O Cocos2D reconhecerá o objeto como uma camada e nos dará os comportamentos de reconhecimento de input que precisamos.

Ao construir um botão, precisamos passar uma imagem. No caso da tela inicial as imagens serão os botões de Play, Highscore e Help, além do botão de som.

Um ponto importante para trabalhar com o evento de toque em um dos botões é habilitar o evento de `Touch` para isso, usamos o comando `this.setIsTouchEnabled(true)`.

```
public class Button extends CCLayer {
    private CCSprite buttonImage;
    private ButtonDelegate delegate;

    public Button(String buttonImage) {
        this.setIsTouchEnabled(true);
        this.buttonImage = CCSprite.sprite(buttonImage);
        addChild(this.buttonImage);
    }
}
```

O que foi feito até aqui é informar ao Cocos2D que teremos objetos do tipo botão, que receberão uma imagem e são habilitados para serem tocados.

O que precisamos agora é:

- Criar os 4 botões: Play, Highscore, Help e Sound.
- Configurar suas posições
- Adicioná-los na tela inicial

Criaremos uma nova classe com a responsabilidade de organizar os botões do Menu. A classe `MenuButtons` herda de `CCLayer` para que possamos mais pra frente detectar eventos de toques.

```
public class MenuButtons extends CCLayer {
    private Button playButton;
    private Button highscoredButton;
    private Button helpButton;
    private Button soundButton;

    public MenuButtons() {
        this.setIsTouchEnabled(true);

        this.playButton = new Button(Assets.PLAY);
        this.highscoredButton = new Button(Assets.HIGHSCORE);
        this.helpButton = new Button(Assets.HELP);
        this.soundButton = new Button(Assets.SOUND);

        // coloca botões na posição correta
        setButtonspPosition();

        addChild(playButton);
        addChild(highscoredButton);
        addChild(helpButton);
        addChild(soundButton);
    }

    private void setButtonspPosition() {
        // Buttons Positions
        playButton.setPosition(
            screenResolution(
                CGPoint.ccp( screenWidth() /2 , screenHeight() - 250 ))
        );

        highscoredButton.setPosition(
            screenResolution(
                CGPoint.ccp( screenWidth() /2 , screenHeight() - 300 ))
        );

        helpButton.setPosition(
            screenResolution(
                CGPoint.ccp( screenWidth() /2 , screenHeight() - 350 ))
        );

        soundButton.setPosition(
```

```
            screenResolution(
                CGPoint.ccp( screenWidth() /2 - 100,
                             screenHeight() - 420 ))
        );
    }
}
```

Falta apenas informar à tela inicial quem é o responsável pelo gerenciamento dos botões. Criaremos então uma instância de `MenuButtons` e adicionaremos nela. Adicione novamente ao construtor da `TitleScreen`:

```
MenuButtons menuLayer = new MenuButtons();
this.addChild(menuLayer);
```

Rode seu aplicativo! Tente clicar nos botões, o que acontece?

Avançamos bem com os botões, posicionando e preparando-os como elementos que podem receber eventos de toque. Porém, não existe ainda um relacionamento entre nossa tela inicial e nosso controle de botões.

Essa é uma parte complexa do desenvolvimento de games. Não é simples coordenar objetos com ciclos de vida diferentes que rodam pela aplicação. O que faremos aqui é utilizar um Design Pattern para auxiliar na comunicação entre os objetos.

Para começar, definiremos um novo tipo no projeto, responsável por garantir que instâncias dele tenham um método que reconhece eventos de tipo em objetos do tipo `Button`. Isso é importante para garantir que apenas receberemos objetos que podem ser analisados e responder ao evento desejado.

```
public interface ButtonDelegate {
    public void buttonClicked(Button sender);
}
```

A classe `MenuButtons` implementará esse comportamento:

```
public class MenuButtons extends CCLayer implements ButtonDelegate {
```

Para complementar, precisamos dar alguma resposta e verificar se os eventos de toque em cada um dos botões está sendo corretamente capturado. Como utilizamos um objeto do tipo `CCLayer` do Cocos2D, podemos sobrescrever o método `buttonClicked`, que envia uma referência do objeto de tipo `Button` que foi pressionado.

Com essa informação em mãos, podemos verificar qual botão foi pressionado e executar o código correspondente. Por enquanto, faremos um log do próprio console para identificar essa interação.

```
@Override
public void buttonClicked(Button sender) {
    if (sender.equals(this.playButton)) {
        System.out.println("Button clicked: Play");
    }

    if (sender.equals(this.highscoredButton)) {
        System.out.println("Button clicked: Highscore");
    }

    if (sender.equals(this.helpButton)) {
        System.out.println("Button clicked: Help");
    }

    if (sender.equals(this.soundButton)) {
        System.out.println("Button clicked: Sound");
    }
}
```

Mas como fazer com que os botões invoquem o método `buttonClicked` do nosso listener, o `MenuButtons`? Vamos criar um método na classe `Button` que define quem deverá ser avisado:

```
public void setDelegate(ButtonDelegate sender) {
    this.delegate = sender;
}
```

Agora que o botão sabe quem deve avisar, devemos fazer com que o `delegate` seja avisado quando esse botão (que é um `CCLayer`) for tocado.

Primeiro devemos sobrescrever o método `registerWithTouchDispatcher` para dizer que o próprio objeto botão receba as notificações de toque:

```
@Override
protected void registerWithTouchDispatcher() {
    CCTouchDispatcher.sharedDispatcher()
            .addTargetedDelegate(this, 0, false);
}
```

Agora, toda vez que alguém tocar nesse layer, receberemos uma invocação para o método `ccTouchesBegan`. Precisamos reescrevê-lo e verificar se o menu em questão foi tocado. Faremos isso através do método `CGRect.containsPoint`, que verificará se o local tocado (`touchLocation`) está dentro da "caixa" que a imagem do menu forma (`buttonImage.getBoudingBox()`):

```
@Override
public boolean ccTouchesBegan(MotionEvent event) {
    CGPoint touchLocation = CGPoint.make(event.getX(), event.getY());
    touchLocation = CCDirector.sharedDirector()
                              .convertToGL(touchLocation);
    touchLocation = this.convertToNodeSpace(touchLocation);

    // Verifica toque no botão
    if (CGRect.containsPoint(
            this.buttonImage.getBoudingBox(), touchLocation)) {
        delegate.buttonClicked(this);
    }

    return true;
}
```

Só falta adicionarmos, na classe `MenuButtons`, a informação de que o delegate dos botões deve avisar a própria classe `MenuButtons`, já que é ela que tem o método `buttonClicked`:

```
public MenuButtons() {
    this.playButton.setDelegate(this);
    this.highscoredButton.setDelegate(this);
    this.helpButton.setDelegate(this);
    this.soundButton.setDelegate(this);
}
```

Rode o projeto e confira os botões recebendo os inputs no console.

Aqui criamos nossa própria classe de `Button`, mas poderíamos ter usado outras classes já prontas do Cocos2D, como `Menu` e `MenuItem`. Ter criado uma `Button` nos ajudará nos capítulos seguintes, no qual teremos botões dentro do próprio jogo, além de termos aprendido um pouco mais dos detalhes internos do framework.

4.8 Conclusão

O jogo deve estar como mostrado na tela abaixo, com background e logo configurados. Além disso, 4 botões foram implementados, o Play, Highscore, Help e controle de Som.

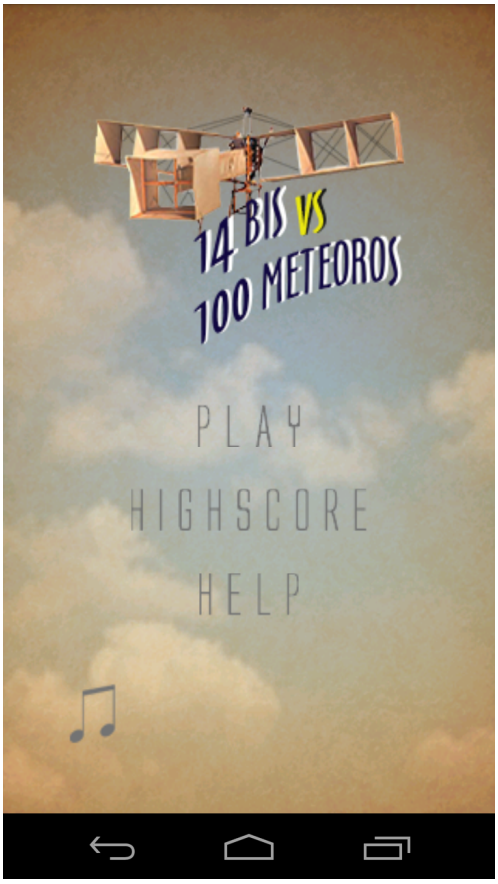


Figura 4.7: Tela de abertura.

O core do desenvolvimento de um jogo não é fácil. Repare na evolução do protótipo para o jogo real que estamos fazendo e perceberá que partes complexas foram encapsuladas pelo Cocos2D.

Nesse capítulo, fomos um pouco mais a fundo em questões como telas(`Scenes`) e camadas(`Layers`). Utilizamos também um importante elemento do Cocos2D, o `Director`.

A seguir, faremos a transição para a tela do jogo e teremos nossos primeiros elementos do game.

Capítulo 5

Tela do jogo e objetos inimigos

Hora de adicionar perigo ao nosso game! Nesse capítulo iremos entrar na tela do jogo de fato, onde toda a ação ocorrerá. Essa será a principal parte do jogo e por isso trataremos em alguns capítulos.

Para iniciar, passaremos pela transição da tela de abertura para a tela de jogo. Além disso, colocaremos os inimigos aparecendo na tela. Alguns conceitos importantes do Cocos2D serão utilizados nesse capítulo, cujo objetivo é ter a tela do game rodando, com alguns inimigos surgindo.

Utilizaremos muito do que já foi visto até aqui, como `CCLayers` para representar camadas de uma tela, `CCSprites` para controlar objetos e `CCScene` para criar a tela do jogo. Além disso o `CCDirector` será utilizado novamente.

No fim desse capítulo teremos a transição entre tela de abertura e tela do game, além dos objetos inimigos aparecendo na tela.

Figura 5.1: Meteoros inimigos.

5.1 GameScene

Precisamos de uma tela para o jogo, para conter os elementos principais de interação do game como player, inimigos e controles. Assim como anteriormente, criaremos uma tela herdando da classe `CCLayer` do Cocos2D, para que possamos ter diversas camadas atuantes, como botões, inimigos, player, score, etc.

Também como anteriormente, a definição de uma tela é criada através de um `CCScene`, que saberá lidar com as camadas da nossa classe.

O Maestro

Idealmente, essa classe não deve ter muitas responsabilidades, mas sim, funcionar como um orquestrador de todos os elementos, ou seja, um maestro em uma orquestra, que dirige e comanda o que todos os outros elementos fazem e como eles interagem entre si.

Ela será a classe que inicializa objetos no jogo, que coloca objetos na tela, porém o comportamento de cada um deles será representado individualmente em cada classe correspondente.

Algumas das responsabilidades da `GameScene`, a classe maestro do jogo, devem ser:

- Iniciar a tela do game e organizar as camadas
- Adicionar objetos como player, inimigos e botões a essas camadas
- Inicializar cada um desses objetos
- Checar colisões entre objetos

A classe `GameScene` tem muita responsabilidade, porém não detém regras e lógicas de cada elemento. Outra função importante dessa classe é aplicar um dos conceitos vistos anteriormente, do *game loop*.

Vamos então criar a classe `GameScene` já colocando um background como fizemos anteriormente na tela de abertura:

```
public class GameScene extends CCLayer {
    private ScreenBackground background;

    private GameScene() {
        this.background = new ScreenBackground(Assets.BACKGROUND);
        this.background.setPosition(
            screenResolution(
            CGPoint.ccp(screenWidth() / 2.0f, screenHeight() / 2.0f)));
        this.addChild(this.background);
    }

    public static CCScene createGame() {
        CCScene scene = CCScene.node();
        GameScene layer = new GameScene();
```

```
        scene.addChild(layer);
        return scene;
    }
}
```

5.2 Transição de telas

Para que o jogo comece, precisamos fazer o link entre a tela de abertura e a tela do game!

Aqui utilizaremos o `CCDirector` que sabe manter uma `CCScene` ativa por vez. Além de trocar de uma tela para outra, o Cocos2D nos permite escolher e configurar detalhes dessa transição.

Utilizaremos o método `replaceScene` que fará uma transição com o tempo de pausa entre uma tela e outro, gerando um efeito suave.

Para isso, na classe `MenuButtons`, mudamos o `buttonClicked` para que o `if` do botão de play comece o jogo:

```
@Override
    public void buttonClicked(Button sender) {
        if (sender.equals(this.playButton)) {
            System.out.println("Button clicked: Play");
            CCDirector.sharedDirector().replaceScene(
                    CCFadeTransition.transition(1.0f,
                                                GameScene.createGame()));
        }
        //...
    }
}
```

Rode o jogo e clique no menu de play. O que acontece? Por enquanto, só temos a tela de background!

5.3 Engines

Temos a classe que orquestrará os objetos do game, e criaremos agora classes responsáveis por gerenciar outros elementos. O primeiro elemento que teremos serão os inimigos. Nossos inimigos serão meteoros que cairão e precisarão ser destruídos pelo player.

Criaremos uma nova camada, um novo layer para representar esses inimigos. Como utilizado anteriormente, camadas são representadas por heranças ao `CCLayer` do Cocos2D.

Para manter o link entre a tela principal e essa camada, utilizaremos o mesmo conceito de `delegates` visto anteriormente, ou seja, a tela to jogo e a camada dos meteoros inimigos serão linkados através dessa delegação, para serem chamadas quando precisarem fazer entre si.

Engine de objetos inimigos

Nossa camada de objetos inimigos, os meteoros, será responsável por criar inimigos e enviar à tela do jogo. Essa engine de meteoros não é responsável pelo movimento do meteoro em si, mas sim de controlar a aparição deles na tela e fazer o link entre objeto `Meteoro` e tela do Game.

É importante que uma `Engine` saiba quando é o momento de colocar um novo elemento no jogo. Muitas vezes, principalmente para objetos inimigos, utilizamos números randômicos para definir a hora de colocar um novo inimigo na tela.

Essa ideia foi muito utilizada por jogos em que o nível mais difícil era apenas uma equação na qual o número randômico gerado satisfazia uma condição de entrada em um `if`. No código da nossa engine abaixo, faremos exatamente isso.

Vale citar que a engine é o código responsável por manter o loop de objetos, e com o Cocos2D, utilizamos métodos de agendamento para isso. Ou seja, criaremos um `schedule` para que a engine analise se deve ou não atualizar e incluir um novo objeto inimigo na tela.

Abaixo, o código da primeira Engine do game, a classe `MeteorsEngine`.

```
public class MeteorsEngine extends CCLayer {
    private MeteorsEngineDelegate delegate;

    public MeteorsEngine() {
        this.schedule("meteorsEngine", 1.0f / 10f);
    }

    public void meteorsEngine(float dt) {
        // sorte: 1 em 30 gera um novo meteoro!
        if(new Random().nextInt(30) == 0){
            this.getDelegate().createMeteor(
                new Meteor(Assets.METEOR));
        }
```

```
    }

    public void setDelegate(MeteorsEngineDelegate delegate) {
        this.delegate = delegate;
    }

    public MeteorsEngineDelegate getDelegate() {
        return delegate;
    }
}
```

Para fechar o link entre ambas as camadas, criaremos uma interface que obrigará a tela do jogo a ter um método para receber os objetos criados por essa `Engine` e colocá-los na tela.

```
public interface MeteorsEngineDelegate {
    public void createMeteor(
            Meteor meteor, float x, float y, float vel,
            double ang, int vl);
}
```

Implemente a interface `MeteorsEngineDelegate` na classe `GameScene` e crie o método que será responsável pelos meteoros que criaremos em seguida. Aproveitaremos para fazer a camada dos meteoros na tela do game.

Mantendo as referências

Um outro ponto importante para o controle do jogo e toda a orquestração é manter todos os objetos criados de uma forma fácil para que possam ser analisados depois. Um exemplo nesse caso é a comparação com o tiro ou com o próprio player para detectar colisões, que serão tratados mais à frente.

Vamos guardar a referência de cada meteoro criado em uma coleção na classe `GameScene`:

```
public class GameScene extends CCLayer implements MeteorsEngineDelegate {
    private ScreenBackground background;
    private MeteorsEngine meteorsEngine;
    private CCLayer meteorsLayer;
    private List meteorsArray;

    @Override
```

```
    public void createMeteor(Meteor meteor, float x, float y, float vel,
        this.meteorsLayer.addChild(meteor);
        meteor.start();
        this.meteorsArray.add(meteor);
    }
}
```

5.4 Meteor

Chegamos ao objeto inimigo propriamente dito. As principais responsabilidades desse objeto são:

- Carregar imagem (Sprite)
- Posicionar elemento na tela
- Guardar a posição do objeto para que possa ser movimentado com o tempo

Esse é o primeiro objeto de jogo realmente que criaremos. Até o momento, criamos telas preenchendo com elementos, botões de menu e classes de engine para dar a base a esses elementos principais do jogo.

A primeira coisa a se fazer é adicionar o asset do meteoro na classe `Assets`.

```
public class Assets {
    public static String  METEOR = "meteor.png";
    // outros assets
}
```

Vamos criar a classe `Meteor`. Inicialmente, cada meteoro nasce no topo da tela (é o valor de `screenHeight`), e numa posição `x` randômica:

```
public class Meteor extends CCSprite {
    private float x, y;

    public Meteor(String image) {
        super(image);
        x = new Random().nextInt(Math.round(screenWidth()));
        y = screenHeight();
    }
}
```

Repare que o objeto meteoro permanece vivo na memória por um bom tempo. Ele é criado e, a cada frame, renderizado em uma posição diferente, dando a impressão de movimento.

Aqui mais uma vez o framework nos ajuda. Para que cada frame seja renderizado durante o jogo, e a posição do objeto mude com o passar do tempo, o Cocos2D nos permite escolher um método que será invocado de tempo em tempo. Isso será definido no `start`, fazendo `schedule("nomeDoMetodo")`. No nosso caso, o método será `update`:

```
public void start() {
    this.schedule("update");
}

public void update(float dt) {
    y -= 1;
    this.setPosition(screenResolution(CGPoint.ccp(x, y )));
}
```

> **Variável dt dos updates**
>
> O Cocos2D vai **tentar** invocar o seu método `update` de x em x milissegundos, isso é, em cada frame. Mas, por uma série de motivos, o processador pode estar ocupado com outras coisas, fazendo com que essa chamada agendada não ocorra quando você queria. Ele pode demorar mais. Nesse caso, vai dar uma impressão que seu jogo está lento, já que a tela será renderizada como se só tivesse passado o tempo de um frame, mas na verdade pode ter passado alguns segundos.
>
> Num jogo profissional, você deve guardar essa informação para decidir corretamente quantos pixels cada objeto deve mudar. No nosso caso, se o `dt` for maior que o de 1 frame, deveríamos descer o meteoro mais que 1 pixel, fazendo a regra de 3.

5.5 Tela do game

Para fechar e linkar a classe da tela do jogo, com a engine de meteoros e com os objetos meteoros criados, modificaremos a classe `GameScene`.

Primeiramente, vamos criar um método que conterá a inicialização dos objetos de jogo. Crie o método `addGameObjects`:

```
private void addGameObjects() {
    this.meteorsArray = new ArrayList();
    this.meteorsEngine = new MeteorsEngine();
}
```

No construtor, criaremos um layer especialmente para os meteoros e adicionaremos a tela do jogo via `addChild`. Além disso, invocaremos o `addGameObjetos`. Nosso construtor ficará assim:

```
private GameScene() {
        this.background = new ScreenBackground(Assets.BACKGROUND);
        this.background.setPosition(
            screenResolution(CGPoint.ccp(screenWidth() / 2.0f,
                                screenHeight() / 2.0f)));
        this.addChild(this.background);

        this.meteorsLayer = CCLayer.node();
        this.addChild(this.meteorsLayer);

        this.addGameObjects();
    }
}
```

Um método importante que utilizaremos aqui é o `onEnter()`. Ele é invocado pelo Cocos2D assim que a tela do game está pronta para orquestrar os objetos do jogo. Ele será a porta de entrada do jogo. Por enquanto simplesmente adicionaremos o `meteorsEngine` e setaremos o `delegate` como `this`, para sermos avisados dos novos meteoros:

```
@Override
public void onEnter() {
    super.onEnter();
    this.startEngines();
}

private void startEngines() {
    this.addChild(this.meteorsEngine);
    this.meteorsEngine.setDelegate(this);
}
```

5.6 Conclusão

Temos agora a tela de abertura e a tela de jogo, onde o game loop é rodado, em funcionamento. Nosso game loop inicializa os inimigos. Precisamos de um player para jogar contra eles, é o que veremos a seguir!

Figura 5.2: Meteoros inimigos.

Capítulo 6

Criando o Player

Tela do jogo preparada e inimigos aparecendo! Cenário perfeito para iniciarmos o desenvolvimento do player! Essa é uma parte bem interessante no desenvolvimento de jogos, pois programaremos o objeto que será controlado pelos inputs do usuário.

Para isso, utilizaremos a maioria dos elementos do framework `Cocos2d` que vimos até agora para trabalhar com o player. Utilizaremos camadas, sprites e os conceitos vistos anteriormente.

Nossa tela de jogo precisará de mais uma camada, utilizaremos `Sprites` para o Player e detectaremos inputs do usuário para movê-lo.

Resumidamente, nesse capítulo faremos:

- Colocar o player na tela
- Movimentar o player
- E atirar!

Daremos um passo importante na construção do jogo nesse capítulo, o objetivo final é ter a cena a seguir, ainda sem detectar colisões.

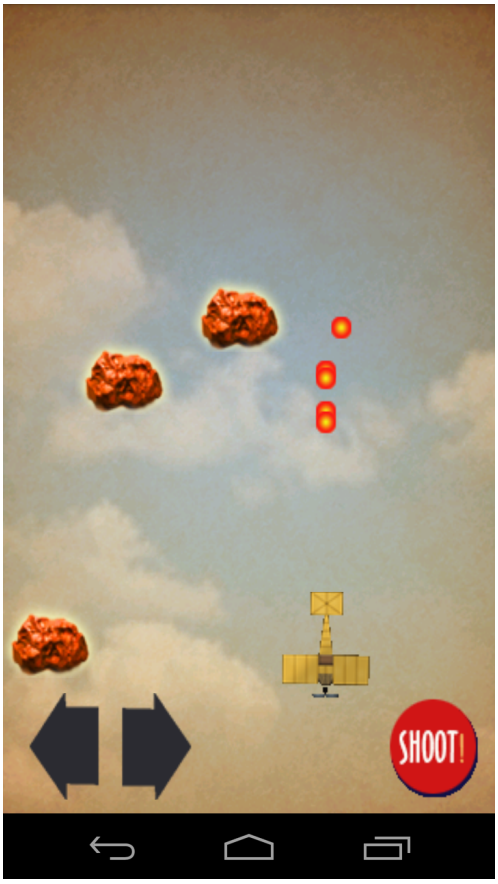


Figura 6.1: 14 bis atirando contra os meteoros.

6.1 Desenhando o Player

Iniciaremos pela imagem, adicionando a figura do player na classe `Assets`.

```
public class Assets {
    public static String NAVE = "nave.png";
}
```

Criaremos o objeto principal e, como anteriormente, controlamos figuras e imagens herdando do `CCSprite` do `Cocos2D`.

Utilizaremos o método que retorna a largura da tela para centralizar o Player. Precisamos de variáveis que guardem essas posições pois precisaremos alterá-las mais à frente.

Como já utilizado pelas outras classes, manteremos o link entre tela de abertura e player utilizando um delegate.

A classe `Player` será iniciada da maneira a seguir.

```
public class Player extends CCSprite{

    float positionX = screenWidth()/2;
    float positionY = 50;

    public Player(){
        super(Assets.NAVE);
        setPosition(positionX, positionY);
    }

    public void setDelegate(ShootEngineDelegate delegate) {
        this.delegate = delegate;
    }

}
```

O objeto Player já está pronto para ser inicializado, mas ainda não existe uma camada no tale do jogo responsável por mostrá-lo. Para que o player apareça na tela do jogo, temos que adicionar mais uma camada. Essa camada terá o nome de `playerLayer`.

Na classe `GameScene` é necessário adicionar a variável de layer, e iniciá-la no construtor. Após isso, adicione a camada criada através do método `addGameObjects`.

```
private CCLayer playerLayer;
private Player player;

private GameScene(){
    this.playerLayer = CCLayer.node();
    this.addChild(this.playerLayer);
}
```

```
private void addGameObjects() {
    this.player = new Player();
    this.playerLayer.addChild(this.player);
}
```

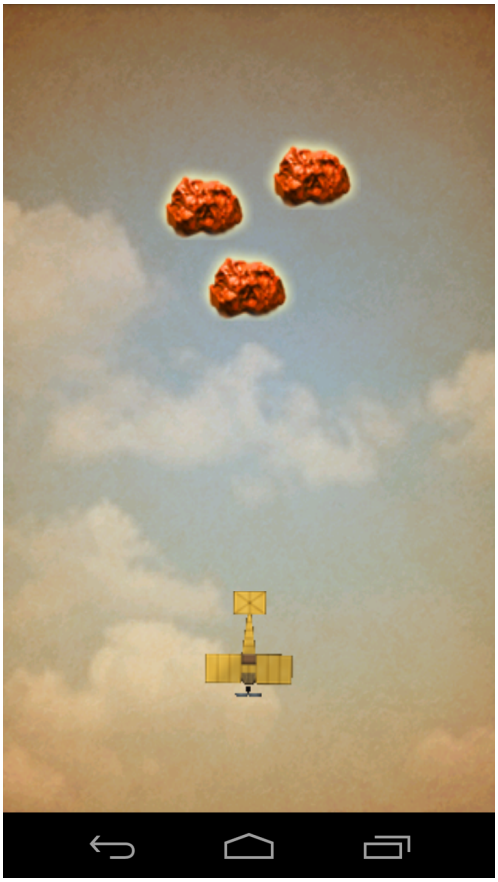


Figura 6.2: 14 bis pronto para ação.

6.2 Botões de controle

Já temos o player aparecendo na tela. Além disso, ele já está em uma camada da tela do game, o que faz com que seja renderizado durante o jogo.

Vamos agora adicionar outros elementos à tela, para que o player possa ser comandado pelos inputs do usuário. Para isso, precisaremos de novas imagens para

esses controles e uma nova classe responsável por essa camada no jogo.

Iniciaremos adicionando 3 imagens, duas para movimentar o player entre direita e esquerda e outra que será o botão de atirar. Essas imagens serão incluídas no arquivo `Assets`.

```
public class Assets {
    public static String LEFTCONTROL = "left.png";
    public static String RIGHTBUTTON = "right.png";
    public static String SHOOTBUTTON = "shootButton.png";
}
```

A classe de botões de controle não é complexa, porém tem algumas características importantes a serem ressaltadas.

Essa classe deve ser tratada como mais uma camada na tela de jogo, por isso, será utilizada como um `CCLayer`. Além disso, essa camada deve executar ações quando receber o toque na tela, ou seja, precisamos de um código que identifique o input do usuário e execute alguma lógica do jogo. Para isso, utilizaremos novamente a interface `ButtonDelegate`.

Começaremos a classe `GameButtons` definindo 3 botões como variáveis do tipo `Button`. No construtor habilitaremos o toque na tela, inicializando os botões, e mantendo o link com a tela de jogo, pelo delegate. Por fim, adicionaremos os botões criados na tela de jogo.

```
public class GameButtons extends CCLayer implements ButtonDelegate {

    private Button leftControl;
    private Button rightControl;
    private Button shootButton;

    public static GameButtons gameButtons() {
        return new GameButtons();
    }

    public GameButtons() {

        // Habilita o toque na tela
        this.setIsTouchEnabled(true);

        // Cria os botões
        this.leftControl     = new Button(Assets.LEFTCONTROL);
```

```
        this.rightControl     =
                new Button(Assets.RIGHTCONTROL);
        this.shootButton      = new Button(Assets.SHOOTBUTTON);

        // Configura as delegações
        this.leftControl.setDelegate(this);
        this.rightControl.setDelegate(this);
        this.shootButton.setDelegate(this);

        // Adiciona os botões na tela
        addChild(leftControl);
        addChild(rightControl);
        addChild(shootButton);

    }

}
```

Não esqueça de adicionar o delegate, fazendo o link entre as telas: Na `GameScene` adicione o código abaixo no construtor.

```
gameButtonsLayer.setDelegate(this);
```

Quase tudo pronto para que os controles apareçam na tela do jogo, porém, ainda é necessário configurar o posicionamento deles. Cada objeto do tipo Button tem um método `setPosition()` que utilizaremos para essa função.

Como temos 3 botões para configurar posicionamentos, separaremos em um método a parte chamado `setButtonspPosition()`. Faremos a chamada a esse método no construtor, antes de adicioná-los na tela.

```
public GameButtons() {

    //...

    // Configura posições
    setButtonspPosition();

    // Adiciona botões na tela
    addChild(leftControl);
    addChild(rightControl);
    addChild(shootButton);
```

```
}

private void setButtonspPosition() {

    // Posição dos botões
    leftControl.setPosition(screenResolution(
                                CGPoint.ccp( 40  , 40 ))) ;
    rightControl.setPosition(screenResolution(
                                CGPoint.ccp( 100 , 40 ))) ;
    shootButton.setPosition(screenResolution(
                        CGPoint.ccp( screenWidth() -40 , 40 )));

}
```

Nesse momento os controles já estão na tela do jogo. Repare que o desenvolvimento de jogos no `Cocos2D` vai sendo em função sempre de `Sprites` e `Camadas` até aqui. A partir de agora entraremos mais na questão de responder aos input com mudanças na tela. É como no protótipo criado no início do livro, no qual movimentamos o player a partir do touch. Porém nesse caso, o código será um pouco mais aprofundado.

Para preparar esse cenário, criaremos um método `buttonClicked`. Para esse momento, apenas faremos o log da direção recebida pelo input e do do clique no botão de atirar.

```
@Override
public void buttonClicked(Button sender) {

    if (sender.equals(this.leftControl)) {
        System.out.println("Button clicked: Left");
    }

    if (sender.equals(this.rightControl)) {
        System.out.println("Button clicked: Right");
    }

    if (sender.equals(this.shootButton)) {
        System.out.println("Button clicked: Shooting!");
    }

}
```

Por fim, é necessário adicionar essa camada de botões à tela do jogo, ou seja, na classe `GameScene`.

```
private GameScene(){
    GameButtons gameButtonsLayer = GameButtons.gameButtons();
    this.addChild(gameButtonsLayer);
    this.addChild(gameButtonsLayer);
}
```

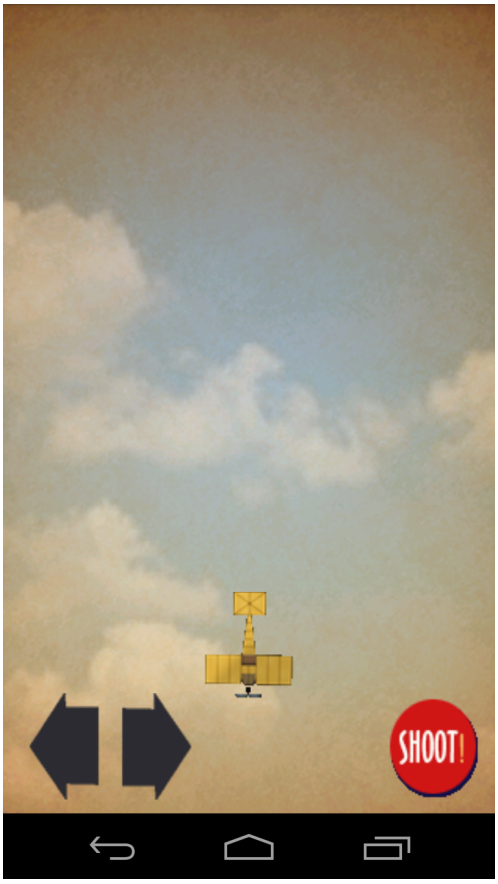


Figura 6.3: Controles de direção e tiro.

Imagens posicionadas e preparadas. Hora de começar a ação!

6.3 Atirando

Falaremos agora de uma parte do jogo que pode parecer simples a princípio, mas tem impacto em muitas partes do jogo para que funcione, o tiro! Para que a nave atire, precisaremos de uma série de coisas, portanto, vamos primeiro listar o que será necessário para organizar o pensamento antes de ir para o código.

- Um novo asset, ou seja, imagem do tiro
- Uma classe que represente o tiro como um `Sprite`
- Definir o posicionamento do tiro na tela
- Uma engine responsável por criar um tiro
- Uma camada na tela do jogo para os tiros
- Associar o tiro e o player, para que o tiro saia do player

Há muita coisa para que o tiro de fato aconteça, porém, se listarmos cada uma dessas dependências podemos simplificar o desenvolvimento. Iniciaremos adicionando a figura do tiro na classe `Assets`.

```
public class Assets {
    public static String SHOOT = "shoot.png";
}
```

Podemos iniciar a programação do tiro. Antes de pensar em fazer a nave atirar ou algo assim, vamos tentar pensar em programação o que é o tiro. O tiro é um *sprite*, ou seja, uma imagem, que anda pela tela de baixo para cima. Ou seja, uma vez que um tiro aparece na tela, precisamos movimentá-lo para cima com o passar do tempo.

Para gerar essa sensação de movimento e controlar esses *updates* do posicionamento do tiro no eixo vertical, criaremos um método com o nome `update{}`. Esse método será executado pelo `Cocos2D` a cada interação. O framework manda como parâmetro um tempo de execução, para que possa ser analisado se algo deve ser ou não executado desde a última vez que ele invocou esse método. No nosso caso, esse parâmetro não será utilizado, pois a lógica do tiro é pelo toque no botão e não por uma regra de tempo.

A classe de tiro precisa manter o link com a tela de jogo, então utilizaremos o *delegate*. Criaremos também um método chamado `start()`, que será utilizado para verificar que o tiro está funcionando.

```
public class Shoot extends CCSprite{

    private ShootEngineDelegate delegate;

    float positionX, positionY;

    public Shoot(float positionX, float positionY){
        super(Assets.SHOOT);
        this.positionX = positionX;
        this.positionY = positionY;
        setPosition(this.positionX, this.positionY);
        this.schedule("update");
    }

    public void update(float dt) {
        positionY += 2;
        this.setPosition(screenResolution(
                CGPoint.ccp(positionX, positionY )));
    }

    public void setDelegate(ShootEngineDelegate delegate) {
        this.delegate = delegate;
    }

    public void start() {
        System.out.println("shoot moving!");

    }
}
```

Assim como fizemos com os meteoros, teremos uma classe responsável por colocar o tiro no jogo. Essa responsabilidade não é da tela de jogo e nem do próprio tiro. A classe `ShootEngineDelegate` fará esse papel. Para isso, criaremos uma interface que diz o que uma engine de tiro deve saber fazer. Nesse caso, criar os tiros.

```
public interface ShootEngineDelegate {
    public void createShoot(
            Shoot shoot);
}
```

Tiro e tela de jogo

Até aqui, a classe de tiro foi definida. Agora vamos à tela do jogo para adicionar esse novo elemento. Duas coisas são necessárias nesse momento. A primeira é a camada do `Cocos2D` para que os tiros apareçam. A segunda é um array que guardará os tiros para que a detecção da colisão com os meteoros seja feita.

```
private CCLayer shootsLayer;
private ArrayList shootsArray;
```

Além de criar as variáveis, é necessário inicializá-las. No construtor criaremos a camada.

```
private GameScene(){
    this.shootsLayer = CCLayer.node();
    this.addChild(this.shootsLayer);

    this.setIsTouchEnabled(true);
}
```

E no método `addGameObjects()` criaremos o array. Aproveitaremos esse momento para fechar o link do `delegate` entre tiro e tela de jogo.

```
private void addGameObjects() {
    this.shootsArray = new ArrayList();
    this.player.setDelegate(this);
}
```

Atirando!

já temos a classe do tiro e também a preparação na tela de jogo para o link entre esses dois objetos. O que fizemos até aqui foi preparar a infra para que o tiro aconteça. Vamos nos concentrar agora na relação entre o tiro e o Player.

Nesse momento, implementaremos a interface `ShootEngineDelegate` na `GameScene`. Dessa forma, a tela de jogo saberá o que deve fazer quando for requisitada para atirar. A interface obriga a criação do método `createShoot()`. Nele, um novo tiro, que é recebido como parâmetro, é adicionado à camada e ao array de tiros. Além disso, chama o método `start()` da classe `Shoot`, permitindo que ela controle o que for necessário lá dentro.

```
@Override
public void createShoot(Shoot shoot) {
```

```
    this.shootsLayer.addChild(shoot);
    shoot.setDelegate(this);
    shoot.start();
    this.shootsArray.add(shoot);
}
```

Criaremos também, ainda na `GameScene`, um método chamado `shoot()`, que chama o player. Precisamos disso por um fato muito importante, que é o posicionamento inicial do tiro. Lembre-se que o tiro deve sair da nave, portanto o Player e seu posicionamento são muito importantes. O método `shoot` é um método da classe `Player`, que contém todas as variáveis de posicionamento da nave na hora do tiro.

```
public boolean shoot() {
    player.shoot();
    return true;
}
```

Apertando o botão!

A classe `GameButtons` é a camada que recebe o input do usuário, ou seja, o momento em que o botão é pressionado, e consequentemente o tiro disparado.

Ela precisa ter um link com a tela de jogo, para que após um toque no botão ser identificado uma ação ocorra. Nesse caso o botão chamará método `shoot()` da tela de jogo, que se encarregará de dar sequência ao tiro.

O código do link entre camada de botões e tela de jogo fica como a seguir.

```
private GameScene delegate;

public void setDelegate(GameScene gameScene) {
    this.delegate = gameScene;

}
```

O código que detecta o toque no botão de tiro e chama o disparo fica assim:

```
if (sender.equals(this.shootButton)) {
    this.delegate.shoot();
}
```

Player atirando

Tudo pronto para atirar a partir do botão de tiro pressionado! Temos a classe de tiro definida, o botão de tiro programado e a tela de jogo com a camada e métodos necessários prontos. precisamos que alguém dê o comando de atirar e nada melhor que o próprio Player para fazer isso! O Player terá o método `shoot()` que captura o posicionamento da nave e chama a *engine* de criação do tiro. Bang!

```
public class Player extends CCSprite{

    private ShootEngineDelegate delegate;

    float positionX = screenWidth()/2;
    float positionY = 50;

    public Player(){
        super(Assets.NAVE);
        setPosition(positionX, positionY);
    }

    public void shoot() {
        delegate.createShoot(
                new Shoot(positionX, positionY));
    }

    public void setDelegate(ShootEngineDelegate delegate) {
        this.delegate = delegate;
    }

}
```

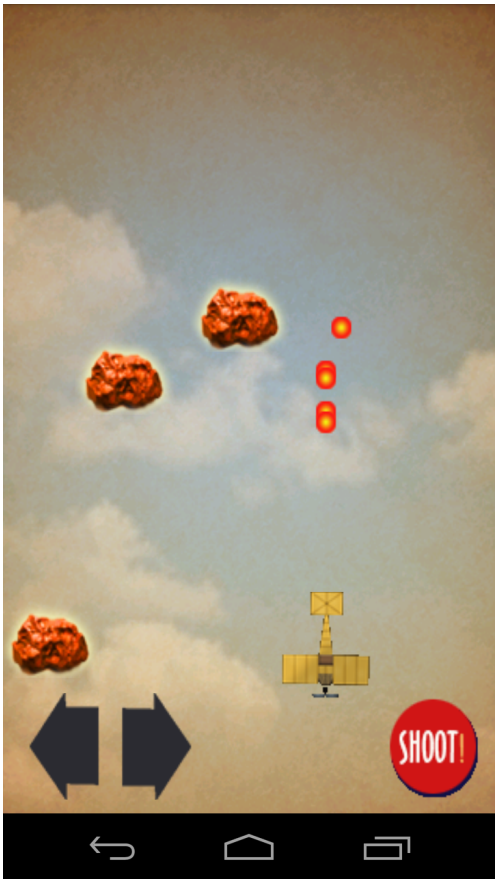


Figura 6.4: Player atirando.

6.4 Movendo o player

Vamos fechar esse capítulo com uma das partes mais importantes do jogo. Após atirar, moveremos o Player para esquerda e direita. Utilizaremos a mesma estratégia de atirar, mas agora as coisas serão mais simples.

Para iniciar, o `Player` deve saber se mover. Movimentar o player é fazer com que sua posição Y seja atualizada quando um evento for detectado.

Na classe `Player.java` adicionaremos dois novos métodos, que quando chamados, mudam a posição horizontal da nave.

```
public void moveLeft() {
```

```
    if(positionX > 30){
        positionX -= 10;
    }
    setPosition(positionX, positionY);
}

public void moveRight() {
    if(positionX <screenWidth() - 30){
        positionX += 10;
    }
    setPosition(positionX, positionY);
}
```

Na `GameScene` criaremos métodos para que essas ações sejam chamadas pelos botões.

```
public void moveLeft() {
    player.moveLeft();
}

public void moveRight() {
    player.moveRight();
}
```

E por fim, na `GameButtons`, invocamos os métodos para que a posição da nave seja alterada.

```
@Override
public void buttonClicked(Button sender) {
    if (sender.equals(this.leftControl)) {
        this.delegate.moveLeft();
    }

    if (sender.equals(this.rightControl)) {
        this.delegate.moveRight();
    }

    if (sender.equals(this.shootButton)) {
        this.delegate.shoot();
    }
}
```

Ao rodar o projeto, devemos ter a nave se movimentando a partir dos toques no comando de controle da nave. Além disso ela já atira de acordo com a posição do Player.

6.5 Conclusão

Esse é um capítulo muito importante para o desenvolvimento do jogo. Além de usar diversos elementos do framework `Cocos2D`, como camadas, sprites e agendamento(*update*), diversos conceitos de jogos foram usados, além de práticas como delegates e engines.

O jogo deve estar como na figura abaixo.

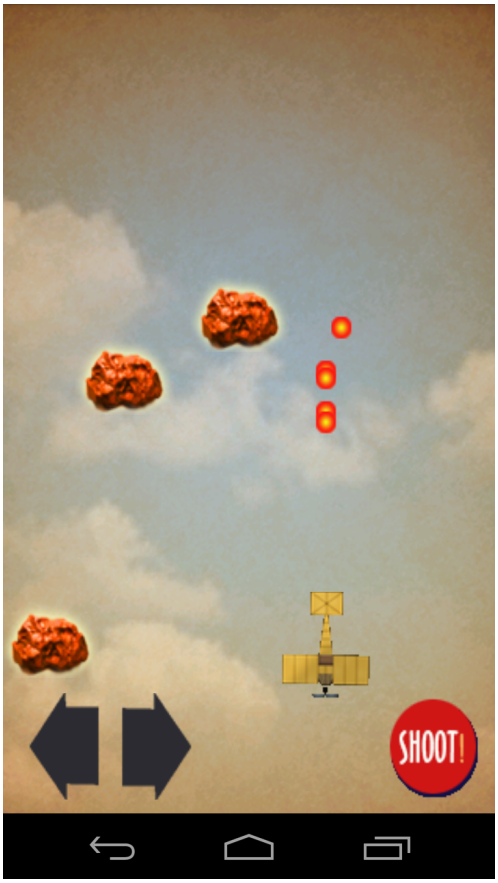


Figura 6.5: 14 bis atirando contra os meteoros.

É hora de verificar as colisões!

Capítulo 7

Detectando colisões, pontuando e criando efeitos

Colisões são o coração dos games. Seja um soco de um personagem no outro, seja o personagem principal capturando algum elemento, ou como no jogo que estamos programando, um tiro que atinge um meteoro.

Detectar que um elemento encostou em outro é um assunto que pode ser muito complexo. Como mostrado no capítulo do protótipo que desenvolvemos, podemos detectar colisões considerando figuras geométricas em volta dos elementos para facilitar.

No capítulo atual, detectaremos colisões em duas situações.

- Quando um tiro atinge um meteoro
- Quando um meteoro atinge o avião (game over)

Veremos aqui que mais uma vez utilizar um framework de desenvolvimento de jogos como o `Cocos2D` ajuda muito nesse trabalho.

Uma vez detectadas as colisões, utilizaremos os efeitos do `Cocos2D` para gerar uma animação quando a detecção ocorrer. Em outra parte importante desse capítulo, falaremos sobre a atualização do placar. Essa deve ser uma parte tranquila pois utilizará conceitos já vistos anteriormente, como *camadas* e *sprites*.

Esse capítulo trará novos códigos e mais utilização do framework `Cocos2D`, sendo um capítulo chave para o desenvolvimento do jogo.

7.1 Detectando colisões

A primeira coisa que precisamos para identificar a colisão entre objetos no game é definir uma estratégia para isso. Nesse jogo, a estratégia será analisar um grupo de objetos em um array, com cada objeto de um outro grupo. Ou seja, dados dois arrays, verificar se algum elemento de um array sobrepõe outro.

Criaremos um array para o player que, inicialmente, terá um único jogador.

```
private List playersArray;
```

E adicionaremos esse player no array de objetos. Mais pra frente você poderá evoluir o jogo e controlar mais de um player.

```
private void addGameObjects() {
    this.playersArray = new ArrayList();
    this.playersArray.add(this.player);
}
```

Definindo as bordas

Precisamos agora de uma forma de definir as bordas ou limites do tiro, da nave e dos meteoros, para que seja possível fazer a detecção da colisão.

No protótipo, utilizamos a estratégia de círculos. Para o jogo atual, utilizaremos uma estratégia de quadrados ou retângulos.

Como estamos utilizando o `Cocos2D` e seus `Sprites` podemos mais facilmente conseguir essas informações.

Criaremos um método que receberá um `Sprite` e devolverá um retângulo, que conterá as bordas do elemento. Para isso, utilizaremos um método que existe nos próprios *Sprites* chamado `getBoundingBox()`. Esse método devolve um tipo `CGRect`, também do `Cocos2D`, que representa os contornos da figura mapeados em forma retangular.

Para trabalhar com essa informação, precisaremos também saber a posição do elemento na tela, e então utilizaremos outro tipo chamado `CGPoint`.

Com esse método, teremos todas as coordenadas do posicionamento do objeto a ser analisado.

```
public CGRect getBoarders(CCSprite object){
    CGRect rect = object.getBoundingBox();
    CGPoint GLpoint = rect.origin;
    CGRect GLrect = CGRect.make(GLpoint.x, GLpoint.y, rect.size.width,
                                rect.size.height);

    return GLrect;
}
```

Checando a colisão

Uma vez que já conseguimos os valores de um elemento do jogo, suas bordas e posição na tela, podemos utilizar a estratégia que falamos no começo do capítulo para identificar se um elemento colide com outro durante o jogo.

Criaremos um método importante para checar as colisões, que precisa de alguns parâmetros para funcionar. Os dois primeiros são arrays de objetos a serem verificados. Ou seja, se queremos checar se os tiros estão colidindo com os meteoros, passaremos esses dois arrays. Outro ponto importante é passar uma referência da tela de jogo, no caso a `GameSCene`. precisamos disso para poder executar algum método caso a colisão seja detectada, porém como diversas colisões podem ser detectas, como nave com tiro ou tiro com meteoro, isso será decidido em tempo de execução. É exatamente por isso que precisamos, por último, de um parâmetro a mais, que recebe o nome do método a ser executado no caso de a colisão acontecer.

O que teremos então é uma verificação de cada elemento do primeiro array, com cada elemento do segundo array. Caso a detecção aconteça, faremos um log por enquanto, e mostraremos quais elementos tiveram a colisão.

Na `GameScene` teremos o código a seguir.

```
private boolean checkRadiusHitsOfArray(List<? extends CCSprite> array1,
    List<? extends CCSprite> array2, GameScene gameScene, String hit) {

    boolean result = false;

    for (int i = 0; i < array1.size(); i++) {
```

```
            // Pega objeto do primeiro array
            CGRect rect1 = getBoarders(array1.get(i));

            for (int j = 0; j < array2.size(); j++) {
                // Pega objeto do segundo array
                CGRect rect2 = getBoarders(array2.get(j));

                // Verifica colisão
                if (CGRect.intersects(rect1, rect2)) {
                    System.out.println("Colision Detected: " + hit);
                    result = true;

                }
            }
        }
        return result;
    }
```

Tendo o método que identifica a colisão como fizemos, podemos utilizá-lo para a verificação de dois arrays quaisquer. Faremos a chamada a ele para detectar colisões entre tiros e meteoros e entre meteoros e a nave. Esse método deve ser chamado de tempos em tempos pelo `Cocos2D`, portanto, utilizaremos mais uma vez o agendamento do framework, passando esse método para o *schedule*.

Ainda na `GameScene` faremos essas chamadas.

```
public void checkHits(float dt) {

    this.checkRadiusHitsOfArray(this.meteorsArray,
            this.shootsArray, this, "meteoroHit");

    this.checkRadiusHitsOfArray(this.meteorsArray,
            this.playersArray, this, "playerHit");
}
```

Como citado anteriormente, agendaremos o método de checagem de colisões. Para isso, o método *schedule* receberá o método *checkHits* como parâmetro.

```
public void onEnter() {
    super.onEnter();
    this.schedule("checkHits");
    this.startEngines();
}
```

Rode o jogo agora repare no console (LogCat) do Eclipse. A colisão já está sendo detectada, e portanto, podemos partir para atualizar o placar e fazer os efeitos necessários!

7.2 Efeitos

No momento que detectamos que dois objetos do jogo colidiram, algumas coisas devem ser feitas. Vamos listar para facilitar o fluxo a seguir.

- Executar uma animação, como explosão ou similares
- Remover os elementos dos arrays
- Atualizar o placar ou mostrar tela de game over

Vamos começar pela animação. Existem diversas possibilidades de animação utilizando o `Cocos2D`. O framework oferece alguns efeitos tradicionais como *fade* e *scale*. A ideia é configurar uma série de ações, que juntas e em um espaço curto de tempo, criam uma animação.

Para o primeiro exemplo, vamos animar o meteoro quando é atingido pelo tiro. Nesse momento, animaremos o meteoro para que fique pequeno, dando a impressão que sumiu por ser atingido.

O que faremos abaixo é:

- Reduzir a escala de tamanho da imagem
- Retirar da tela com um efeito leve, chamado *fade out*
- Rodar ambas em sequência

Após rodar a sequência de animações, precisamos retirar do array o meteoro da memória, pois ele não existe mais. Precisamos também parar o agendamento desse objeto, que roda de tempos em tempos atualizando a posição e gerando o movimento.

Primeiramente, vamos alterar nossa interface `MeteorsEngineDelegate` para que ela receba a notificação de quando um meteoro colidiu:

```
public interface MeteorsEngineDelegate {
    public void createMeteor(
            Meteor meteor, float x, float y, float vel,
            double ang, int vl);

    public void removeMeteor(Meteor meteor);
}
```

Na nossa classe `Meteor`, definimos que ela poderá ter um delegate, que será avisado das colisões:

```
public class Meteor extends CCSprite {
    private MeteorsEngineDelegate delegate;

    public void setDelegate(MeteorsEngineDelegate delegate) {
        this.delegate = delegate;
    }
    // restante do código
}
```

Agora, na nossa `GameScene`, toda vez que criarmos um novo `Meteor`, queremos que ele avise a própria `GameScene` de que ele foi removido. Em outras palavras, a `GameScene` será o delegate do `Meteor`. Altere o `createMeteor` para que ele invoque o `setDelegate`:

```
@Override
public void createMeteor(Meteor meteor, float x, float y, float vel,
        double ang, int vl) {
    meteor.setDelegate(this);
    this.meteorsLayer.addChild(meteor);
    meteor.start();
    this.meteorsArray.add(meteor);
}
```

Teremos um novo método na `Meteor`, que será invocado quando houver a colisão:

```
public void shooted() {
    this.delegate.removeMeteor(this);

    // para de ficar chamando o update
    this.unschedule("update");
```

```
    float dt = 0.2f;
    CCScaleBy a1 = CCScaleBy.action(dt, 0.5f);
    CCFadeOut a2 = CCFadeOut.action(dt);
    CCSpawn s1 = CCSpawn.actions(a1, a2);

    CCCallFunc c1 = CCCallFunc.action(this, "removeMe");
    this.runAction(CCSequence.actions(s1, c1));
}
```

No método acima retiramos o link entre o objeto e a tela de jogo, cancelamos o agendamento da atualização de posição e criamos as ações que juntas farão o efeito de sumir do meteoro.

O `Cocos2D` possui ainda um método que pode ser invocado para que o objeto seja liberado e coletado. Esse método é o `removeFromParentAndCleanup()` e será invocado logo após a animação acabar. Fazemos isso via `CCCallFunc` pois queremos que o método `removeMe` seja invocado depois de a animação terminar.

```
public void removeMe() {
    this.removeFromParentAndCleanup(true);
}
```

Animando o tiro

Utilizaremos o mesmo pensamento para animar o tiro quando colidir com um meteoro. Para isso, criaremos o método `explode()` na classe `Shoot.java`, alterando apenas alguns parâmetros.

Esse método terá que executar algumas ações:

- Retirar o objeto meteoro do array
- Parar o agendamento da movimentação do meteoro
- Animar a explosão do meteoro
- Limpar o objeto da memória

Faremos isso no método a seguir:

```
public void explode() {
    // Remove do array
```

```
        this.delegate.removeShoot(this);

        // Para o agendamento
        this.unschedule("update");

        // Cria efeitos
        float dt = 0.2f;
        CCScaleBy a1 = CCScaleBy.action(dt, 2f);
        CCFadeOut a2 = CCFadeOut.action(dt);
        CCSpawn s1 = CCSpawn.actions(a1, a2);

        // Função a ser executada após efeito
        CCCallFunc c1 = CCCallFunc.action(this, "removeMe");

        // Roda efeito
        this.runAction(CCSequence.actions(s1, c1));

}
```

Para limpar o objeto da memória, chamaremos novamente o método do `Cocos2D` chamado `removeFromParentAndCleanup`. Repare que ele foi chamado no código acima, na forma de string.

```
public void removeMe() {
    this.removeFromParentAndCleanup(true);
}
```

Já temos o código que o meteoro deve executar quando uma colisão for detectada. Agora precisamos fazer a chamada a ele nos momento de colisão. Voltando à classe `GameScene` temos o método `checkRadiusHitsOfArray`, que percorre dois arrays e verifica intersecções.

Nesse método, quando dois objetos estiverem colidindo, algo deve ser feito. Essa é uma parte importante do jogo e precisa de muito cuidado. Repare que o nosso método de detecção é genérico, ou seja, ele recebe dois arrays de objetos e analisa se ocorre colisão entre eles. No momento que essa colisão é detectada, ele precisará executar algo que pode ser a explosão da nave, ou a explosão de um meteoro.

É importante perceber que a decisão de qual método rodar após detectada a colisão é em tempo de execução, e precisaremos invocá-lo via `reflection`.

Na classe `GameScene` altere o método `checkRadiusHitsOfArray`:

```
if (CGRect.intersects(rect1, rect2)) {
    System.out.println("Colision Detected: " + hit);
    result = true;

    Method method;
    try {
        method = GameScene.class.getMethod(hit,
                CCSprite.class, CCSprite.class);

        method.invoke(gameScene, array1.get(i),
                array2.get(j));

    } catch (SecurityException e1) {
        e1.printStackTrace();
    } catch (NoSuchMethodException e1) {
        e1.printStackTrace();
    } catch (IllegalArgumentException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    }
}
```

A partir daqui, basta fazer as chamadas aos métodos `shooted` e `explode`. Na classe `GameScene` adicione o método `meteoroHit()`:

```
public void meteoroHit(CCSprite meteor, CCSprite shoot) {
    ((Meteor) meteor).shooted();
    ((Shoot) shoot).explode();
}
```

Rode o projeto e teste o que fizemos até aqui!

Removendo objetos

Precisamos lembrar que embora os objetos não estejam mais aparecendo na tela eles continuam alocados na memória, ou seja, ainda não foram removidos. Criaremos agora o código que eliminará os objetos dos arrays após colisão entre tiro e meteoro.

Iniciaremos adicionando duas declarações nas interfaces. A primeira será a `MeteorsEngineDelegate`, onde adicionaremos o método `removeMeteor`:

```
public interface MeteorsEngineDelegate {
    public void createMeteor(
            Meteor meteor, float x, float y, float vel, double ang,
            int vl);

    public void removeMeteor(Meteor meteor);
}
```

E a segunda é a interface `ShootEngineDelegate` que terá a definição do `removeShoot`:

```
public interface ShootEngineDelegate {
    public void createShoot(
            Shoot shoot);

    public void removeShoot(Shoot shoot);
}
```

Feito isso, basta fazer a remoção dos objetos na classe `GameScene`, criando os métodos `removeMeteor` e `removeShoot`:

```
@Override
public void removeMeteor(Meteor meteor) {
    this.meteorsArray.remove(meteor);

}

@Override
public void removeShoot(Shoot shoot) {
    this.shootsArray.remove(shoot);
}
```

7.3 Player morre

Utilizando a mesma estratégia vamos animar o player quando um meteoro colidir com ele. Para isso, na classe `Player` teremos o método `explode` como abaixo:

```
public void explode() {
    // Para o agendamento
```

```
    this.unschedule("update");

    // Cria efeitos
    float dt = 0.2f;
    CCScaleBy a1 = CCScaleBy.action(dt, 2f);
    CCFadeOut a2 = CCFadeOut.action(dt);
    CCSpawn s1 = CCSpawn.actions(a1, a2);

    // Roda os efeitos
    this.runAction(CCSequence.actions(s1));
}
```

Para que seja executado, crie o método `playerHit` na classe `GameScene`:

```
public void playerHit(CCSprite meteor, CCSprite player) {
    ((Meteor) meteor).shooted();
    ((Player) player).explode();
}
```

Rode o projeto e verifique a tela do jogo!

7.4 Placar

Para fechar esse capítulo vamos adicionar uma pontuação para cada meteoro destruído após ser atingido por um tiro. Utilizaremos alguns conceitos já vistos anteriormente.

Primeiramente, trataremos o valor de pontos que aparece na tela como uma nova camada, e camadas para o `Cocos2D` são classes que herdam de `CCLayer`.

Essa camada apresentará os pontos em número para o jogador. para isso, precisamos de duas variáveis. A primeira é do tipo inteiro, que guarda os pontos atuais. A segunda é um campo de texto para colocar esse valor na tela.

Além disso, veremos como utilizar um tipo de letra (*font type*) diferente para fazer isso. Existe no `Cocos2D` uma classe chamada `CCBitmapFontAtlas`. Essa classe possui um método chamado `bitmapFontAtlas` que recebe uma string e uma fonte a ser usada.

Depois disso, basta configurar o tamanho da letra e posicionamento. Crie a classe `Score`.

```
public class Score  extends CCLayer {
    private int score;
```

```
    private CCBitmapFontAtlas text;

    public Score(){
        this.score = 0;

        this.text = CCBitmapFontAtlas.bitmapFontAtlas(
                String.valueOf(this.score),
                "UniSansSemiBold_Numbers_240.fnt");

        this.text.setScale((float) 240 / 240);

        this.setPosition(screenWidth()-50, screenHeight()-50);
        this.addChild(this.text);
    }
}
```

Alteraremos o valor da variável `score` com um método chamado `increase`. Esse método poderá ser chamado sempre que uma colisão entre tiro e meteoro for detectada.

Seu código é simples, apenas incrementa a variável `score` e configura novamente o texto do placar.

```
public void increase() {
    score++;
    this.text.setString(String.valueOf(this.score));
}
```

Invoque o `increase` toda vez que um meteoro for atingindo, modificando o `meteoroHit`:

```
public void meteoroHit(CCSprite meteor, CCSprite shoot) {
    ((Meteor) meteor).shooted();
    ((Shoot) shoot).explode();
    this.score.increase();
}
```

Agora que temos a camada do placar preparada, vamos adicioná-la à tela principal. Para isso, criaremos dois objetos, um do tipo `CCLayer` e outro do tipo `Score`.

Na classe `GameScene` adicione:

```
private CCLayer scoreLayer;
private Score score;
```

É necessário iniciar a camada e adicioná-la a tela. No construtor da classe `GameScene` adicione:

```
this.scoreLayer = CCLayer.node();
this.addChild(this.scoreLayer);
```

Faça a chamada ao método `increase` ná última linha do método `meteoroHit`, na classe `GameScrene`:

```
public void meteoroHit(CCSprite meteor, CCSprite shoot) {
    ((Meteor) meteor).shooted();
    ((Shoot) shoot).explode();

    this.score.increase();

}
```

Para finalizar, basta criar o objeto do tipo `Score` e adicionar a camada correspondente.

Ainda na classe `GameScene` adicione essa chamada ao método `addGameObjects`:

```
private void addGameObjects() {
    //...

    // placar
    this.score = new Score();
    this.scoreLayer.addChild(this.score);
}
```

7.5 Conclusão

Detectar colisões de forma manual, como feito no capítulo do protótipo, não é tão simples e envolve muitos cálculos matemáticos. Porém, utilizando um framework como o `Cocos2D` as coisas são facilitadas.

Nesse capítulo passamos pelo que pode ser considerado o coração do jogo, a detecção de colisões. A partir delas, executamos efeitos e atualizamos a tela para o jogador.

O próximo capítulo tratará de uma parte muito importante para dar vida aos jogos, os sons e efeitos.

Capítulo 8

Adicionando sons e música

Os sons são de fundamental importância no desenvolvimento de um game. Hoje existem profissões como *sound designers* que trabalham especificamente criando os sons dos games. Muitos jogos utilizam orquestras para executar sua trilha sonora.

A música dá vida ao jogo, torna-o mais divertido e dá respostas ao jogador para as partes importantes.

Existem duas formas principais de sons no mundo dos games: música e efeitos.

Quando o jogo começa uma música de fundo normalmente dá o clima do jogo. Essa música é normalmente executada em background e se repete inúmeras vezes ao longo do game. Além dela, existem os efeitos de som gerados em momentos importantes, como quando uma colisão é detectada ou quando o placar é alterado

Para o nosso jogo, utilizaremos sons encontrados gratuitamente no site http://www.freesound.org/. Você pode buscar diversos tipos de sons nesse site para o seu próximo game!

8.1 Executando sons

Nessa primeira etapa utilizaremos o framework `Cocos2D` para adicionar som a 3 eventos do jogo. Utilizaremos 3 arquivos de sons diferentes:

- Disparo de um tiro
- Colisão do tiro com um meteoro
- Colisão entre meteoro e avião

Podemos utilizar os formatos mais comuns para adicionar sons ao nosso jogo, e aqui o formato escolhido será `wav`. Colocaremos os sons no diretório `res/raw` do nosso projeto.

Você pode encontrar todos os sons que serão utilizados nesse capítulo nesse link: https://github.com/andersonleite/jogos_android_14bis/tree/master/res/raw

SoundEngine

Para lidar com sons, o `Cocos2D` disponibiliza uma classe chamada `SoundEngine`. Essa classe possui diversos métodos para trabalhar com sons e música no game. Para trabalhar com essa classe não é necessário criar uma instância mas sim, utilizar um *Singleton* disponibilizado pelo framework. Para isso executamos `SoundEngine.sharedEngine()` tendo acesso às opções de sons de que precisamos.

Com esse acesso, podemos executar músicas e sons, parar e iniciar arquivos de áudio, aumentar e diminuir o volume etc. Nesse momento, iniciaremos executando 3 sons utilizando o método `playEffect()`.

Esse método recebe como parâmetro a referência ao arquivo de áudio do diretório `res/raw` que pode ser acessada utilizando o `CCDirector`.

O primeiro efeito de som, o tiro, será colocado na classe `Shoot`. Adicione o código abaixo ao método `start()`:

```
public void start() {
    SoundEngine.sharedEngine().playEffect(
        CCDirector.sharedDirector().getActivity(), R.raw.shoot);
}
```

O próximo som será executado quando da colisão entre meteoro e tiro. Na classe `Meteor` adicione o código abaixo ao método `shooted`:

```
public void shooted() {
    SoundEngine.sharedEngine().playEffect(
        CCDirector.sharedDirector().getActivity(), R.raw.bang);

    // ...
}
```

Para finalizar, adicionaremos um som quando um meteoro atingir o avião. Na classe `Player` adicione o código abaixo ao método `explode`:

```
public void explode() {
    SoundEngine.sharedEngine().playEffect(
        CCDirector.sharedDirector().getActivity(), R.raw.over);

    // ...
}
```

Agora o game já executará os sons quando um dos 3 eventos acima, tiro e colisões, acontecerem!

8.2 Cache de sons

Vimos como executar sons utilizando o `Cocos2D`. Porém, você deve ter reparado em um problema grave. A primeira vez que um dos sons precisa ser tocado, ele demora muito. Isso se deve ao fato do framework inicializar o som apenas no momento que foi necessário. Existe uma estratégia e boa prática para solucionar esse problema, o cache de sons.

Para colocar um som no cache devemos iniciá-lo já no início do game. Criaremos então um método com essa responsabilidade na classe `GameScene` chamado `preloadCache`. Esse método fará o cache dos 3 sons que estamos utilizando até aqui.

```
private void preloadCache() {
    SoundEngine.sharedEngine().preloadEffect(
            CCDirector.sharedDirector().getActivity(),
            R.raw.shoot);

    SoundEngine.sharedEngine().preloadEffect(
            CCDirector.sharedDirector().getActivity(),
            R.raw.bang);
```

```
    SoundEngine.sharedEngine().preloadEffect(
            CCDirector.sharedDirector().getActivity(),
            R.raw.over);
}
```

Adicione a chamada do método `preloadCache` no construtor da `GameScene`, na última linha do método:

```
preloadCache();
```

Rode novamente o projeto e verifique que agora os sons respondem perfeitamente ao momento dos tiros e colisões!

8.3 Música de fundo

Agora que já trabalhamos com os sons do game, vamos ver como lidar com a música do jogo. A música será iniciada quando o jogador entrar na tela de jogo. Para isso, usaremos a mesma classe `SoundEngine` porém com o método `playSound`.

No construtor da `GameScene` adicione a chamada a música.

```
SoundEngine.sharedEngine().playSound(
                CCDirector.sharedDirector().getActivity(),
                R.raw.music, true);
```

Outro método importante é o método que para a música. Esse método é chamado `pauseSound()`. Faremos isso logo após o efeito de explosão entre meteoro e avião na classe `Player`:

```
public void explode() {
    SoundEngine.sharedEngine().playEffect(
        CCDirector.sharedDirector().getActivity(), R.raw.over);
    SoundEngine.sharedEngine().pauseSound();
}
```

Rode o projeto e veja como o som adiciona vida ao jogo. Tiro, colisões e música devem estar funcionando nesse momento!

8.4 Conclusão

Sons e música são muito importante para os jogos. Repare nos jogos famosos ou mesmo os que você mais gosta e repare na trilha sonora. Mesmo os jogos antigos possuíam sons muito especiais para cada jogo.

É importantíssimo definir uma boa trilha sonora e efeitos para o seu próximo game!

Capítulo 9

Voando com a gravidade!

O mundo dos games foi totalmente revitalizado com os `smartphones`. Jogos que até então eram simples e já não atraiam mais tanto a atenção dos jogadores foram remodelados com as novas possibilidades de experiência que os aparelhos modernos podem proporcionar.

A maioria dos jogos que fazem sucesso nos celulares hoje faz uso de algum recurso do aparelho, como `touch screen`, arrastando objetos na tema, ou utilizando a gravidade com o acelerômetro, e movimentando os elementos de uma maneira diferente dos jogos para consoles.

Nesse capítulo trocaremos a movimentação do avião feita por botões pelo controle movimentando o celular! A experiência será levada a outro nível, não mais sendo um simples jogo com botões, mas sim, utilizando um recurso nativo do aparelho que faz aquele jogo ganhe muita jogabilidade!

Ao fim do capítulo, nosso avião poderá percorrer a tela, ficando como na figura a seguir:

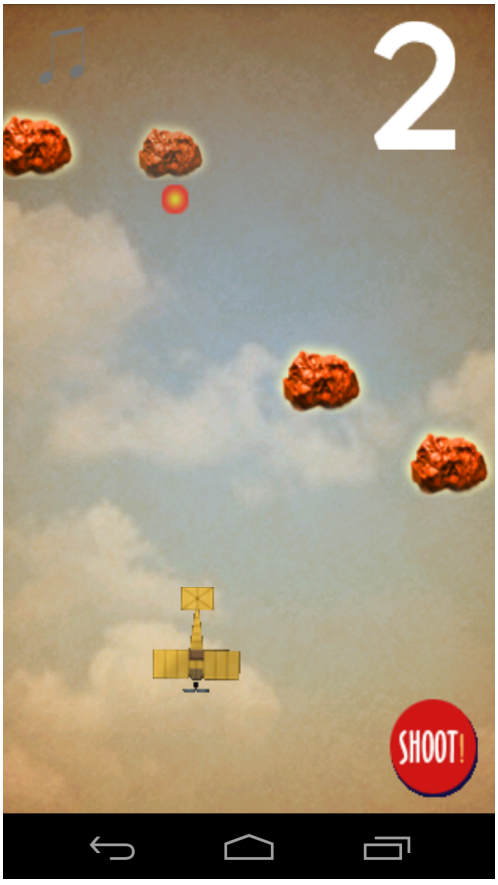


Figura 9.1: Controle por acelerômetro.

9.1 Usando o Acelerômetro

Vamos iniciar planejando o uso do acelerômetro e sabendo onde queremos chegar. Nosso objetivo é retirar os botões que movimentam o avião para esquerda e direita. Sem esses botões, moveremos o avião a partir das coordenadas que o celular captar de movimento no aparelho. Dividiremos esse trabalho em 3 partes principais.

- Capturar as coordenadas de movimentação horizontal e vertical do aparelho
- Controlar a instabilidade do movimento do avião
- Calibrar essas coordenadas para o controle funcionar em posições diferentes

Capturando as coordenadas

A primeira coisa que precisamos saber é como o `Android` pode nos enviar informações do acelerômetro. Isso é feito através da interface `SensorEventListener`. O que ocorre quando uma classe implementa essa interface?

Toda classe que implementa `SensorEventListener` é obrigada a implementar seus métodos, dentre eles o `onSensorChanged(SensorEvent acceleration)`. O que ocorre aqui é que a cada movimentação do device, esse método é chamado pelo `Android`. Repare que ao chamar esse método, o `Android` passa como parâmetro um objeto do tipo `SensorEvent`, que chamaremos de `acceleration`.

Essa é uma parte muito importante, e você deve reparar que com esse objeto que chamamos de `acceleration` temos os valores de posição do aparelho. Esses valores serão informados em 3 variáveis, que representam os eixos X, Y e Z do aparelho, como demostrado na figura a seguir.

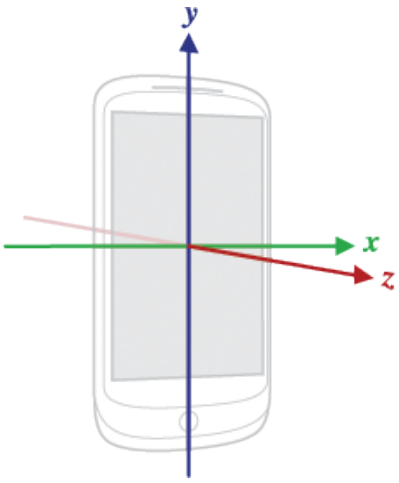


Figura 9.2: Eixos X, Y e Z.

De posse dessas informações, conseguiremos saber se o aparelho se moveu para um determinado lado e atualizar a posição do avião. Essa é uma conclusão importante, ou seja, quando percebemos que o celular está inclinado para um determinado lado, atualizamos essa posição do avião.

A classe Accelerometer

Vamos implementar a classe responsável por capturar esses valores. Mais à frente veremos como fazer o link entre ela e a movimentação do avião.

Para começar criaremos a classe `Accelerometer` que implementa a interface `SensorEventListener`. Essa interface nos obriga a implementar 2 métodos. Teremos o código abaixo:

```
public class Accelerometer implements SensorEventListener {

    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {
        // TODO Auto-generated method stub
    }

    @Override
    public void onSensorChanged(SensorEvent acceleration) {

    }

}
```

Vamos agora ver como capturar as posições X e Y do aparelho. Para isso, criaremos 2 variáveis que vão guardar essa informação e atualizaremos o método `onSensorChanged`:

```
private float currentAccelerationX;
private float currentAccelerationY;

@Override
public void onSensorChanged(SensorEvent acceleration) {
    this.currentAccelerationX = acceleration.values[0];
    this.currentAccelerationY = acceleration.values[1];
}
```

O que temos até aqui? Uma classe que recebe informações de movimentação do aparelho. Ela não foi inicializada ainda, mas logo mais poderá ser utilizada para atualizar a posição da nave.

Para continuar, faremos que essa classe se comporte como um `Singleton`, tendo apenas um objeto do tipo `Accelerometer` no jogo. Para isso, na própria classe `Accelerometer` adicione o seguinte código:

```
static Accelerometer sharedAccelerometer = null;

public static Accelerometer sharedAccelerometer() {
    if (sharedAccelerometer == null) {
        sharedAccelerometer = new Accelerometer();
    }
    return sharedAccelerometer;
}
```

Dessa forma, não corremos riscos de mais de um objetos desse tipo ser instanciado no jogo. Agora, já podemos pensar em fazer o link com a classe `Player`.

Configurando o player

Toda vez que o acelerômetro atualizar as posições devemos mover o avião. Isso ocorrerá em uma quantidade muito grande de vezes por segundo, dando a impressão de movimento. Como em outras classes, fazemos link entre elas utilizando os `delegates`, porém aqui teremos que colocar mais uma coisa. Toda vez que o método `onSensorChanged` da classe `Accelerometer` for executado, ele chamará a classe `Player` informando as coordenadas X e Y.

Para garantir que esse método exista na classe `Player` uma interface será criada. Chamaremos essa interface de `AccelerometerDelegate`.

```
public interface AccelerometerDelegate {
    public void accelerometerDidAccelerate(float x , float y);
}
```

E implementaremos o método `accelerometerDidAccelerate` na classe `Player` para receber as atualizações do acelerômetro. Esse método recebe como parâmetros as coordenadas X e Y, que iremos guardar nas variáveis `currentAccelX` e `currentAccelX`. Repare que ao implementá-lo deixaremos um log para entender o tipo de valor em que as coordenadas são enviadas.

```
public class Player extends CCSprite implements AccelerometerDelegate {
    @Override
    public void accelerometerDidAccelerate(float x, float y) {
        System.out.println("X: " + x);
        System.out.println("Y: " + y);

        // Leitura da aceleracao
```

```
        this.currentAccelX = x;
        this.currentAccelY = y;
    }
}
```

Você pode executar o projeto agora e olhar no console(logcat) as coordenadas sendo impressas.

O que temos neste momento? A classe `Accelerometer` que recebe coordenadas do aparelho e a classe `Player` preparada para receber esses valores e armazená-los em variáveis.

Mas quando a classe `Accelerometer` chama o método `accelerometerDidAccelerate` da classe `Player`? Para que isso ocorra precisamos de duas coisas. A primeira é configurar o delegate entre elas, e então fazer a chamada ao método em questão.

Na classe `Accelerometer` crie o delegate e seu `getter` e `setter`:

```
private AccelerometerDelegate delegate;

public void setDelegate(AccelerometerDelegate delegate) {
    this.delegate = delegate;
}

public AccelerometerDelegate getDelegate() {
    return delegate;
}
```

Na classe `Player` criaremos uma referência ao `Accelerometer`:

```
private Accelerometer accelerometer;
```

Temos a base para fazer o link entre as classes e estamos quase prontos para inicializar o Acelerômetro.

Iniciando o Acelerômetro

Sabemos nesse momento como utilizar o acelerômetro, porém, para terminar o link entre a classe `Accelerometer` que recebe as coordenadas e a classe `Player` que moverá o avião, temos que inicializar o acelerômetro no jogo.

A dificuldade aqui é que o acelerômetro só pode ser inicializado por uma `Activity`, e assim como fizemos para utilizar o `touch screen`, teremos que inicializá-lo através da `MainActivity`.

Nela, inicializaremos o acelerômetro e deixaremos guardada sua referência na classe `DeviceSettings`.

Para isso, na classe `DeviceSettings` o código abaixo:

```
public class DeviceSettings {
    private static SensorManager sensormanager;

    public static void setSensorManager(SensorManager sensormanager) {
        this.sensormanager = sensormanager;
    }

    public static SensorManager getSensormanager() {
        return sensormanager;
    }
}
```

E na `MainActivity` faça a configuração criando o método a seguir:

```
private void configSensorManager() {
    SensorManager sensorManager =
        (SensorManager) getSystemService(Context.SENSOR_SERVICE);
    DeviceSettings.setSensorManager(sensorManager);
}
```

O método `onCreate` da `MainActivity` deve iniciar essa configuração:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // orientacao vertical
    setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_PORTRAIT);
    requestWindowFeature(Window.FEATURE_NO_TITLE);
    getWindow().setFlags(
        WindowManager.LayoutParams.FLAG_FULLSCREEN,
        WindowManager.LayoutParams.FLAG_FULLSCREEN);

    // tela
    CCGLSurfaceView glSurfaceView = new CCGLSurfaceView(this);
    setContentView(glSurfaceView);
    CCDirector.sharedDirector().attachInView(glSurfaceView);
```

```
    // sensor
    configSensorManager();
}
```

Link entre classes

Tendo a classe `Accelerometer`, que captura as coordenadas configuradas, a classe `Player` pronta para receber os valores e o acelerômetro inicializado na `MainActivity`, podemos fazer o link entre elas.

Iniciaremos criando um método que, ao inicializar a classe `Accelerometer` busca a referência ao acelerômetro.

Na classe `Accelerometer` crie o código abaixo:

```
private SensorManager sensorManager;

public Accelerometer() {
    this.catchAccelerometer();
}

public void catchAccelerometer() {
    sensorManager = DeviceSettings.getSensormanager();
    sensorManager.registerListener(this,
            sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER),
            SensorManager.SENSOR_DELAY_GAME);
}
```

Aqui, buscamos o acelerômetro, e deixamos a classe pronta para ser chamada pelo `Android` a cada movimentação do aparelho. Repare que a interface `SensorEventListener` é genérica, ela pode também receber avisos de outros sensores. Nesse caso, pedimos apenas o acelerômetro através do `sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)`. Há diversos outros sensores com os quais você pode adicionar novas funcionalidades aos seus jogos.

Já podemos fazer a chamada à classe `Player`, que é o delegate, e passar as coordenadas. Altere o método `onSensorChanged` na classe `Accelerometer` como a seguir:

```
@Override
public void onSensorChanged(SensorEvent acceleration) {
    this.currentAccelerationX = acceleration.values[0];
```

```
    this.currentAccelerationY = acceleration.values[1];

    // Envia leitura do acelerometro
    if (this.delegate != null) {
        this.delegate.accelerometerDidAccelerate(
                currentAccelerationX, currentAccelerationY);
    }
}
```

Na classe `Player`, adicione o delegate como no código a seguir:

```
public void catchAccelerometer() {
    Accelerometer.sharedAccelerometer().catchAccelerometer();
    this.accelerometer = Accelerometer.sharedAccelerometer();
    this.accelerometer.setDelegate(this);
}
```

Quase tudo pronto. Temos aqui a configuração do `Android` para utilizar o acelerômetro, uma classe responsável por buscar essa referência e conseguir as coordenadas, e o link feito entre a classe `Player` para que possamos movimentar o avião. Mas quem inicializou tudo isso? Ninguém ainda, e utilizaremos nosso `GameScene`, que é a centralizadora, para iniciar essa nova `feature`.

Na classe `GameScene` adicione a chamada ao método `catchAccelerometer` no método `startGame`:

```
public void startGame() {
    // Captura o acelerometro
    player.catchAccelerometer();
}
```

Movendo o player

Agora que já estamos recebendo as coordenadas do acelerômetro e guardando-as em variáveis na classe `Player` já podemos movimentá-lo. O método `update` é que altera a posição do avião, então, vamos alterá-lo.

A ideia aqui será mover uma das quatro possíveis coordenadas do avião, seja horizontal(direita ou esquerda) ou vertical(para cima e para baixo). As coordenadas enviadas pelo acelerômetro seguem o padrão dos eixos, fazendo com que movimentações para um lado sejam números positivos e para o lado oposto número negativo. O mesmo ocorre para o eixo X, enviando números positivos para uma direção e negativos para a posição oposta.

Com base nessa informação, analisaremos as coordenas que o acelerômetro enviou e alteraremos a posição do avião.

Na classe `Player` altere o método `update`:

```
public void update(float dt) {
    if(this.currentAccelX< 0){
        this.positionX++;
    }

    if(this.currentAccelX> 0){
        this.positionX--;
    }

    if(this.currentAccelY< 0){
        this.positionY++;
    }

    if(this.currentAccelY> 0){
        this.positionY--;
    }

    // Configura posicao do aviao
    this.setPosition(CGPoint.ccp(this.positionX, this.positionY));
}
```

Já é possível rodar o jogo e ver o avião se movendo a partir da movimentação do aparelho! Faça o teste.

Experimente deixar a tela paralela a uma mesa. Algo estranho, não? Parece que ele está enfrentando um certa "turbulência".

9.2 Controlando a instabilidade

Você deve ter percebido que o controle do avião ficou instável. Isso ocorre porque não estamos usando nenhuma tolerância para mover o avião, ou seja, movemos sempre independente de ser uma movimentação realmente válida do aparelho. O acelerômetro não é perfeito, além de que ele pega movimentações minúsculas, sempre gerando eventos!

Para melhorar isso, vamos usar um limite. Em vez de comparar com zero, utilizaremos uma constante de tolerância. Chamaremos essa constante de `NOISE`. Ela

definirá o valor mínimo que o acelerômetro deve ser alterado para realmente mover o avião.

Altere o método `update` e cria a constante `NOISE` na classe `Player`.

```
private static final double NOISE = 1;

public void update(float dt) {
    if(this.currentAccelX< -NOISE){
        this.positionX++;
    }

    if(this.currentAccelX> NOISE){
        this.positionX--;
    }

    if(this.currentAccelY< -NOISE){
        this.positionY++;
    }

    if(this.currentAccelY> NOISE){
        this.positionY--;
    }

    // Configura posicao do aviao
    this.setPosition(CGPoint.ccp(this.positionX, this.positionY));
}
```

A partir deste momento, devemos ter um bom controle do avião, podendo movimentá-lo por toda a tela. O único inconveniente é que não temos uma calibração para utilizar o acelerômetro, ou seja, ele funciona bem para posição inicial zero, que é a aquela quando deixamos o aparelho parado em uma mesa, por exemplo.

9.3 Calibrando a partir da posição inicialdo aparelho

Vamos utilizar uma estratégia de calibração no jogo! A ideia é não se basear apenas na posição enviada pelo acelerômetro para mover o player, mas fazer antes algumas contas para entender a posição que o jogador está segurando o aparelho e considerá-la como a posição ZERO, ou posição inicial.

Para isso faremos algumas alterações na classe `Accelerometer`. Criaremos novas variáveis que serão responsáveis por guardar informações sobre a calibração. Essas variáveis guardarão as informações iniciais e se já temos a calibração concluída.

Na classe `Accelerometer` crie as seguintes variáveis:

```
private float calibratedAccelerationX;
private float calibratedAccelerationY;

private int calibrated;
```

Alteraremos também o método `onSensorChanged`. Nele, faremos um loop que receberá as 100 primeiras informações do acelerômetro. Com esses 100 primeiros valores guardaremos a posição inicial do aparelho.

Por que 100 vezes? O acelerômetro demora para enviar as coordenadas corretas do aparelho, e precisamos esperar que ele termine esse trabalho com precisão. Com isso, conseguiremos definir qual a posição inicial do aparelho.

A partir disso, faremos uma alteração no valor que é enviado para mover o avião. Ao invés de enviar o valor diretamente informado pelo acelerômetro, vamos tirar a posição inicial, para ter apenas a mudança relativa àquela movimentação.

Na classe `Accelerometer` o método `onSensorChanged` deve ficar como a seguir:

```
@Override
public void onSensorChanged(SensorEvent acceleration) {
    if(calibrated < 100){
        this.calibratedAccelerationX += acceleration.values[0];
        this.calibratedAccelerationY += acceleration.values[1];

        System.out.println(acceleration.values[0]);
        System.out.println(acceleration.values[1]);

        calibrated++;

        if (calibrated == 100 ) {
            this.calibratedAccelerationX /= 100;
            this.calibratedAccelerationY /= 100;
        }
        return;
    }
```

```
    // Leitura da aceleracao
    this.currentAccelerationX =
        acceleration.values[0] - this.calibratedAccelerationX;

    this.currentAccelerationY =
        acceleration.values[1] - this.calibratedAccelerationY;
}
```

Você pode manter os `logs` por um tempo para entender os valores enviados e depois apagar. Nesse momento o avião já deve estar sendo controlado pela movimentação do aparelho!

Retirando os botões

Provavelmente você não vai querer mover mais o avião utilizando os botões esquerda e direita. Uma forma simples de removê-los é comentar as duas linhas a seguir, na classe `GameButtons`:

```
// addChild(leftControl);
// addChild(rightControl);
```

9.4 Desafios com o acelerômetro

Utilizar recursos como acelerômetro pode tornar o jogo muito mais divertido e engajador, sendo daqueles detalhes que fazem a experiência do game ser totalmente única. Que tal melhorar ainda mais essa experiência com as sugestões abaixo?

- Controle de velocidade: Você pode fazer que, quanto mais inclinado, o avião deslize mais. Se ele estiver pouco inclinado, em vez de incrementar a variável x em 1, você incrementaria em 0.5 ou algo proporcional à aceleração indicada pelo acelerômetro.
- O acelerômetro pode mandar sinais invertidos de acordo com a inclinação. Dependendo da calibração, você precisa detectar se isso esta ocorrendo, para não mudar a orientação do avião repentinamente! Isso dá um certo trabalho...

9.5 Conclusão

Esse é um capítulo trabalhoso, mas muito gratificante. Utilizar recursos dos aparelhos é um dos principais apelos da revolução dos jogos para celular. Saber trabalhar

bem com esses poderosos recursos pode elevar o jogo a um nível de jogabilidade muito mais interessante!

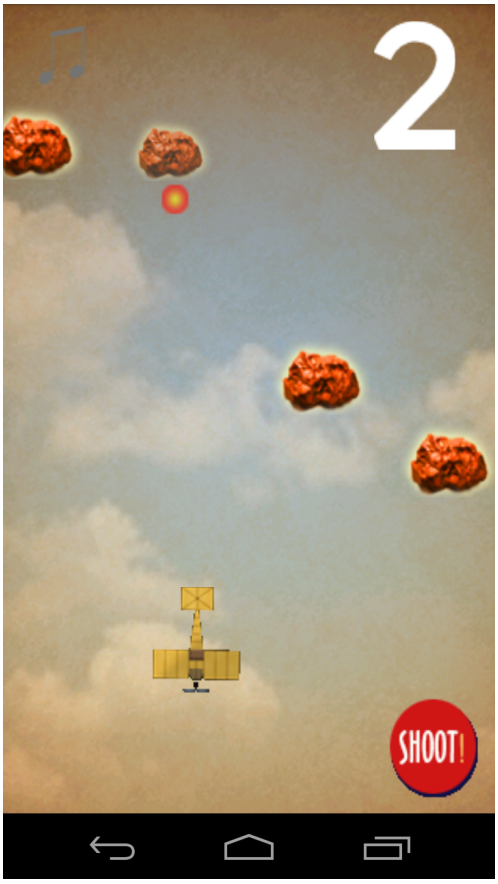


Figura 9.3: Controle por acelerômetro.

Capítulo 10

Tela final e game over

Todo o fluxo do jogo está bem encaminhado, desde a tela de abertura, passando pelas colisões e efeitos, sons e acelerômetro. Agora criaremos duas telas que fecharão o ciclo principal do game.

A primeira tela a ser criada será a tela mostrada quando o jogo acabar, ou seja, quando o 14bis vencer os 100 meteoros.

Essa tela será composta por uma música final, por uma imagem nova e um botão de inicio do jogo.

Ao final, deveremos ter a tela como abaixo.

Figura 10.1: Tela do final do jogo.

10.1 Tela final

Para montar a tela de final de jogo precisaremos de mais uma imagem, a `finalend.png`. Essa imagem deve ficar no diretório `assets` do projeto e será incluída na nossa classe de `Assets`.

```
public class Assets {
    public static String FINALEND = "finalend.png";
}
```

Vamos criar a classe que representará a tela final. Ela será uma nova `CCLayer` e seu nome será `FinalScreen`.

Nela, criaremos uma variável de background e outra para o botão que inicia o game novamente:

```
package com.example.nave.game.scenes;

public class FinalScreen extends CCLayer{
    private ScreenBackground background;
    private Button beginButton;

    public CCScene scene() {
        CCScene scene = CCScene.node();
        scene.addChild(this);
        return scene;
    }
}
```

O construtor dessa classe deve inicializar os objetos *background*, *som* e *imagem de logo*. Faremos isso como já fizemos em outras telas: Crie o construtor da classe `FinalScreen`:

```
public FinalScreen() {
    // background
    this.background = new ScreenBackground(Assets.BACKGROUND);
    this.background.setPosition(
        screenResolution(CGPoint.ccp(screenWidth() / 2.0f,
                        screenHeight() / 2.0f)));
    this.addChild(this.background);

    // som
    SoundEngine.sharedEngine().playSound(
            CCDirector.sharedDirector().getActivity(),
            R.raw.finalend, true);

    // imagem
    CCSprite title = CCSprite.sprite(Assets.FINALEND);
    title.setPosition(
        screenResolution(CGPoint.ccp( screenWidth() /2 ,
                        screenHeight() - 130 ))) ;
    this.addChild(title);
}
```

Configuraremos, ainda no construtor, o botão que inicia o jogo novamente:

```
this.setIsTouchEnabled(true);
this.beginButton = new Button(Assets.PLAY);
this.beginButton.setPosition(
    screenResolution(CGPoint.ccp( screenWidth() /2 ,
                                  screenHeight() - 300 ))) ;
this.beginButton.setDelegate(this);
addChild(this.beginButton);
```

Essa tela terá um botão, portanto, implementaremos a interface `ButtonDelegate` e o método `buttonClicked`.

```
public class FinalScreen extends CCLayer implements ButtonDelegate {

    @Override
    public void buttonClicked(Button sender) {
        if (sender.equals(this.beginButton)) {
            SoundEngine.sharedEngine().pauseSound();

            CCDirector.sharedDirector()
                .replaceScene(new TitleScreen().scene());
        }
    }
```

Para que essa classe possa ser vista no jogo, a classe `GameScene` deve estar ciente e inicializá-la. Para isso, teremos o método `startFinalScreen` que faz a transição para a tela final.

Na classe `GameScene` adicione o seguinte método:

```
public void startFinalScreen() {
    CCDirector.sharedDirector()
              .replaceScene(new FinalScreen().scene());
}
```

E quando teremos a tela de final do jogo? Faremos isso quando 100 meteoros forem destruídos! Porém, para facilitar os testes, mostraremos a tela final quando 5 meteoros forem destruídos.

Na classe `Score` deixe o código do método `increase` como a seguir:

```
public void increase() {
    score++;
    this.text.setString(String.valueOf(this.score));
```

```
    if(score==5){
        this.delegate.startFinalScreen();
    }
}
```

Ao destruir 5 meteoros já devemos ter a tela final com o som da vitória!

Figura 10.2: Tela do final do jogo.

10.2 Tela Game Over

A tela de game over seguirá a mesma lógica da tela final, porém deverá ser inicializada em um outro momento. Quando? Simples, quando um meteoro atingir o

avião!

Vamos ao código. Primeiro, adicionando uma nova figura na classe `Assets`.

```
public class Assets {
    public static String GAMEOVER = "gameover.png";
}
```

Crie a classe `GameOverScreen` que é uma `CCLayer`. Nessa classe, utilizaremos a mesma ideia da tela de final de jogo:

```
package com.example.nave.game.scenes;

public class GameOverScreen extends CCLayer{

    private ScreenBackground background;
    private Button beginButton;

    public CCScene scene() {
        CCScene scene = CCScene.node();
        scene.addChild(this);
        return scene;
    }

}
```

O construtor da classe `GameOverScreen` também é bem simples, parecido com o que já conhecemos:

```
public GameOverScreen() {
    // background
    this.background = new ScreenBackground(Assets.BACKGROUND);
    this.background.setPosition(
        screenResolution(CGPoint.ccp(screenWidth() / 2.0f,
                                     screenHeight() / 2.0f)));
    this.addChild(this.background);

    // image
    CCSprite title = CCSprite.sprite(Assets.GAMEOVER);
    title.setPosition(
        screenResolution(CGPoint.ccp( screenWidth() /2 ,
                                     screenHeight() - 130 ))) ;
    this.addChild(title);
```

```
    // habilita o toque na tela
    this.setIsTouchEnabled(true);
    this.beginButton = new Button(Assets.PLAY);
    this.beginButton.setPosition(
        screenResolution(CGPoint.ccp( screenWidth() /2 ,
                                    screenHeight() - 300 ))) ;
    this.beginButton.setDelegate(this);
    addChild(this.beginButton);
}
```

Para que o botão de reiniciar o jogo funcione, implemente a interface `ButtonDelegate` e o método `buttonClicked` na classe `GameOverScreen`:

```
public class GameOverScreen extends CCLayer
                            implements ButtonDelegate {
    @Override
    public void buttonClicked(Button sender) {
        if (sender.equals(this.beginButton)) {
            SoundEngine.sharedEngine().pauseSound();
            CCDirector.sharedDirector()
                      .replaceScene(new TitleScreen().scene());
        }
    }
}
```

A chamada a essa tela deve ser feita quando a colisão entre meteoro e avião for detectada. Para isso, adicione a transição ao método `playerHit` da `GameScene`:

```
public void playerHit(CCSprite meteor, CCSprite player) {
    ((Meteor) meteor).shooted();
    ((Player) player).explode();
    CCDirector.sharedDirector()
              .replaceScene(new GameOverScreen().scene());
}
```

A tela de game over deve estar aparecendo quando o meteoro colide com o avião, como mostrado a seguir.

Figura 10.3: Tela de game over.

10.3 Conclusão

Esse é um capítulo simples, pois já conhecemos tudo que é necessário para criação de telas e transições. Você pode usar sua imaginação e criar diversas novas telas no game!

Capítulo 11

Pausando o jogo

Nesse capítulo falaremos de mais uma parte importantíssima de um jogo, a tela de pause. Essa tela não costuma ser das mais divertidas de ser desenvolvida, até mesmo pela falsa impressão que pode ser uma tela simples. Porém, tenha atenção aqui! Teremos muitos conceitos importantes nesse momento.

Para não se enganar, vamos à lista de funcionalidades que uma tela de pause deve ter.

- Construir uma nova camada para a tela de pause
- Criar uma classe que entenderá se o jogo está em pause ou rodando
- Criar mais um botão na tela de jogo, o botão *pause*
- Fazer o link entre a tela de pause e tela de jogo
- Parar realmente os objetos na tela

Veja como a tela de pause pode enganar. São muitas coisas a serem feitas para que tudo funcione bem.

Repare que, sempre que possível, fazer uma lista de funcionalidades esperadas na tela pode ajudar a ver com mais profundidade o trabalho que será necessário desenvolver.

Ao final desse capítulo a tela deverá estar como abaixo:

Figura 11.1: Tela de pause.

11.1 Montando a tela de pause

Vamos começar de forma simples e com o que já vimos até o momento. A tela de pause é na verdade mais uma camada dentro de uma tela, ou seja, não mudaremos

de cena (`CCScene`). Como anteriormente, todo nova camada no `Cocos2D` pode ser representada pela classe `CCLayer`.

Uma parte importante dessa tela é que ela terá 2 botões, o continuar, que volta pro jogo, e o sair, que vai para a tela de abertura.

Criaremos os dois botões nessa camada e implementaremos uma interface já utilizada, em outras partes do game, a `ButtonDelegate`.

Crie a classe `PauseScreen` como abaixo:

```
public class PauseScreen extends CCLayer implements ButtonDelegate {
    private Button resumeButton;
    private Button quitButton;
    private CCColorLayer background;
}
```

A próxima etapa é adicionar a seguinte lista nessa tela:

- Habilitar o touch nessa camada
- Definir um background, nesse caso um preto escuro transparente
- Colocar o logo, como na tela de abertura
- Adicionar os botões
- Posicionar os botões

Na classe `Assets` adicione:

```
public static String  EXIT = "exit.png";
```

Faremos as definições acima no construtor da classe. Crie o construtor da `PauseScreen` como abaixo:

```
public PauseScreen() {
    // habilita o toque na tela
    this.setIsTouchEnabled(true);

    // background
    this.background = CCColorLayer.node(ccColor4B.ccc4(0, 0, 0, 175),
            screenWidth(),
            screenHeight());
    this.addChild(this.background);
```

```
    // logo
    CCSprite title = CCSprite.sprite(Assets.LOGO);
    title.setPosition(screenResolution(
        CGPoint.ccp( screenWidth() /2 , screenHeight() - 130 ))) ;
    this.addChild(title);

    // Adiciona botoes
    this.resumeButton = new Button(Assets.PLAY);
    this.quitButton = new Button(Assets.EXIT);
    this.addChild(this.resumeButton);
    this.addChild(this.quitButton);

    // Posiciona botoes
    this.resumeButton.setPosition(screenResolution(
        CGPoint.ccp( screenWidth() /2 , screenHeight() - 250 ))) ;
    this.quitButton.setPosition(screenResolution(
        CGPoint.ccp( screenWidth() /2 , screenHeight() - 300 ))) ;

}
```

Nesse momento temos a tela de pause, porém essa é só a primeira parte. Repare que criar uma tela de pause não é algo diferente das telas anteriores do jogo, porém, fazer o controle que ela precisa demandará novos conceitos. Vamos a eles.

11.2 Controlando o Game Loop

Lembra da analogia de desenhos em blocos de papel do capítulo sobre protótipos? Falamos que um jogo pode ser comparado a uma sequência de imagens que são desenhadas a cada mudança no posicionamento dos objetos do jogo, gerando a impressão de movimento.

O que isso importa para a tela de pause? Precisamos ter um controle de que o jogo está rodando, ou seja, estamos na parte de jogo realmente e não em telas de abertura, por exemplo. Além disso, precisamos saber se o jogo está pausado ou não naquele momento.

Com essas duas variáveis de controle de estado, podemos definir se devemos mostrar a tela de pause, se devemos paralisar os objetos na tela, retomar os movimentos dos objetos etc.

Faremos esse controle com uma classe que terá apenas essa responsabilidade.

Crie a classe `Runner`, como duas variáveis de controle:

```
public class Runner {
    private static boolean isGamePlaying;
    private static boolean isGamePaused;
}
```

Essa classe terá algumas peculiaridades. Uma delas é que ela só terá uma instância ativa no projeto, para que não ocorram confusões entre os estados.

Para isso, o construtor será privado e a instância será controlada pela variável *runner*.

Crie a variável abaixo e o construtor privado na classe `Runner`:

```
static Runner runner = null;

private Runner(){

}
```

Sempre que precisarmos fazer a verificação do estado do game, chamaremos um método que nos devolverá a referência que está cuidado disso. Caso não tenha sido criada ainda, criaremos nesse momento.

Adicione o método `check()` na classe `Runner`:

```
public  static Runner check(){
        if (runner!=null){
            runner = new Runner();
        }
        return runner;
    }
```

Para finalizar a classe de controle, criaremos os *getters* e *setters* das classes da classe de controle.

Crie os métodos abaixo na classe `Runner`:

```
public static boolean isGamePlaying() {
    return isGamePlaying;
}

public static boolean isGamePaused() {
    return isGamePaused;
```

```
}

public static void setGamePlaying(boolean isGamePlaying) {
    Runner.isGamePlaying = isGamePlaying;
}

public static void setGamePaused(boolean isGamePaused) {
    Runner.isGamePaused = isGamePaused;
}
```

Essa classe pode parecer simples. Um *Singleton* com *getters* e *setters*. Porém, sua funcionalidade é de extrema importância no game. Vamos utilizá-las em vários momentos.

11.3 Adicionando o botão de pause

A próxima etapa é relativamente simples. Iremos adicionar o botão pause na tela do jogo. Para isso precisamos de um novo arquivo, a imagem do botão pause. Adicione-o a classe `Assets`.

```
public static String PAUSE    = "pause.png";
```

Adicionaremos esse novo botão na classe `GameButtons`. Aqui, replicaremos o que já temos com os outros botões. O objetivo é criar uma nova variável do tipo `Button`, adicioná-la na camada de botões, configurar o delegate e posicioná-la na tela.

Para fechar, no método `buttonClicked()`, faremos uma chamada ao método `pauseGameAndShowLayer`. Esse método será criado na `GameScene` e fará a chamada à tela de pause.

Deixe a classe `GameButtons` como abaixo:

```
public class GameButtons extends CCLayer implements ButtonDelegate {
    private Button leftControl;
    private Button rightControl;
    private Button shootButton;
    private Button pauseButton;

    private GameScene delegate;

    public static GameButtons gameButtons() {
```

```
        return new GameButtons();
    }

    public GameButtons() {
        // habilita o toque na tela
        this.setIsTouchEnabled(true);

        // Cria os botoes
        this.leftControl     = new Button(Assets.LEFTCONTROL);
        this.rightControl     = new Button(Assets.RIGHTCONTROL);
        this.shootButton     = new Button(Assets.SHOOTBUTTON);
        this.pauseButton     = new Button(Assets.PAUSE);

        // configura delegacoes
        this.leftControl.setDelegate(this);
        this.rightControl.setDelegate(this);
        this.shootButton.setDelegate(this);
        this.pauseButton.setDelegate(this);

        // configura posicoes
        setButtonspPosition();

        // adiciona botoes a tela
        addChild(leftControl);
        addChild(rightControl);
        addChild(shootButton);
        addChild(pauseButton);
    }

    private void setButtonspPosition() {
        // posicao dos botoes

        leftControl.setPosition(
                screenResolution(CGPoint.ccp( 40   , 40 ))) ;
        rightControl.setPosition(
                screenResolution(CGPoint.ccp( 100 , 40 ))) ;
        shootButton.setPosition(
                screenResolution(CGPoint.ccp(screenWidth() -40, 40)));

        pauseButton.setPosition(
                screenResolution(CGPoint.ccp(40, screenHeight() - 30 )));
```

```
    }

    @Override
    public void buttonClicked(Button sender) {
        if (sender.equals(this.leftControl)) {
            this.delegate.moveLeft();
        }

        if (sender.equals(this.rightControl)) {
            this.delegate.moveRight();
        }

        if (sender.equals(this.shootButton)) {
            this.delegate.shoot();
        }

        if (sender.equals(this.pauseButton)) {
            // configuraremos mais a frente
        }
    }

    public void setDelegate(GameScene gameScene) {
        this.delegate = gameScene;
    }
}
```

11.4 A interface entre jogo e pause

De fato uma tela de pause não é tão simples como pode parecer, certo? Recapitulando, nesse capítulo criamos até aqui 3 coisas: A tela de pause, uma classe de controle de estados e adicionamos o botão de pause na tela de jogo.

É sempre bom parar e analisar o que já foi feito em tarefas que são grandes como essas, e envolvem diversas classes.

Tendo a classe da tela de pause e os botões, precisamos fazer o jogo entender quando ele deve mostrá-la.

Nessa jogo, precisamos fazer um link entre a classe principal do jogo, a `GameScene`, e a tela de pause. Criaremos uma interface para dizer o que a tela do jogo deve saber fazer, nesse caso, parar o jogo, continuar e sair.

Crie a interface `PauseDelegate`:

```
public interface PauseDelegate {
    public void resumeGame();

    public void quitGame();

    public void pauseGameAndShowLayer();
}
```

Para preparar a tela de Pause para fazer o link com a tela de jogo, criaremos um delegate. Na `PauseScreen` adicione o código abaixo:

```
private PauseDelegate delegate;

public void setDelegate(PauseDelegate delegate) {
    this.delegate = delegate;
}
```

Adicione os delegates dos botões `resumeButton` e `quitButton` ao construtor da classe `PauseScreen`. Essa parte deve ficar como mostrada a seguir:

```
public PauseScreen() {

    // codigo

    // botoes
    this.resumeButton = new Button(Assets.PLAY);
    this.quitButton = new Button(Assets.EXIT);
    this.resumeButton.setDelegate(this);
    this.quitButton.setDelegate(this);
    this.addChild(this.resumeButton);
    this.addChild(this.quitButton);
}
```

11.5 Pausando o jogo

Começaremos a fazer o *link* entre tudo que foi visto nessa capítulo agora! A principal ideia é orquestrar tudo que fizemos pela classe `GameScene`, que é a tela do jogo e receberá o evento de pause e ativará a `PauseScreen`.

Para que a tela de pause possa ser vista, precisaremos adicionar uma nova camada na `GameScene`. Criaremos uma camada do tipo `CCLayer` e um objeto do tipo `PauseScreen`.

Na `GameScene` crie as variáveis:

```
private PauseScreen pauseScreen;
private CCLayer layerTop;
```

No construtor da `GameScene` iniciaremos a camada e vamos adicionar a mesma à cena atual.

```
private GameScene() {
    // ...

    this.layerTop = CCLayer.node();
    this.addChild(this.layerTop);
}
```

Feito isso, uma nova camada existe na tela de jogo, mas ainda sem relação com a tela de pause que queremos.

Métodos de pause

Vamos seguir fazendo esse link. Faremos a tela de jogo saber que alguns métodos de pause são necessários, nesse caso, *pause*, *resume* e *quit*.

A `GameScreen` implementará a interface que diz o que deve ser feito pelo pause.

```
public class GameScene extends CCLayer implements MeteorsEngineDelegate,
        ShootEngineDelegate, PauseDelegate {
```

Vamos implementar cada um dos 3 métodos de pause agora. O primeiro será o *pauseGame()*. O que esse método deve fazer é verificar se o game está rodando, ou seja, estamos jogando na tela de jogo, e se o jogo não está já em pause. Caso isso seja verdadeiro, configuramos a variável de pause para *true*.

Na `GameScene` crie o método `pauseGame()`:

```
private void pauseGame() {
    if (!Runner.check().isGamePaused() &&
                Runner.check().isGamePlaying()) {

        Runner.setGamePaused(true);

    }
}
```

O próximo passo é um método que remove a tela de pause para continuar o jogo. Nele, configuramos o pause para *false*. É válido chamar o método `setIsTouchEnabled` pois ao voltar das camadas alguns devices perdem essa configuração de *touch*.

Adicione o método `resumeGame` ma `GameScreen`:

```
@Override
public void resumeGame() {
    if (Runner.check().isGamePaused() ||
        ! Runner.check().isGamePlaying()) {

        // Continua o jogo
        this.pauseScreen = null;
        Runner.setGamePaused(false);
        this.setIsTouchEnabled(true);
    }
}
```

Para fechar, implementaremos o método `quit`, o mais simples entre os 3. Nele, Vamos parar os sons que estamos tocando. Além disso, faremos uma transição para a tela de abertura.

```
@Override
    public void quitGame() {
        SoundEngine.sharedEngine().setEffectsVolume(0f);
        SoundEngine.sharedEngine().setSoundVolume(0f);

        CCDirector.sharedDirector()
                  .replaceScene(new TitleScreen().scene());
    }
```

Iniciando tudo

Precisamos configurar algumas variáveis de controle de estado logo no começo do jogo. No método `onEnter()` da `GameScreen` vamos adicionar as configurações de *play* e *pause* abaixo:

```
public void onEnter() {
    super.onEnter();

    // Configura o status do jogo
```

```
    Runner.check().setGamePlaying(true);
    Runner.check().setGamePaused(false);

    // pause
    SoundEngine.sharedEngine().setEffectsVolume(1f);
    SoundEngine.sharedEngine().setSoundVolume(1f);

    // verifica colisoes
    this.schedule("checkHits");

    this.startEngines();
}
```

Criaremos agora um método responsável por iniciar a tela de pause, que será acionado pelo botão de pause. Primeiramente, crie o método abaixo na classe `GameScreen`:

```
public void pauseGameAndShowLayer() {
    if (Runner.check().isGamePlaying() &&
        ! Runner.check().isGamePaused()) {

        this.pauseGame();
    }

    if (Runner.check().isGamePaused() &&
            Runner.check().isGamePlaying() &&
            this.pauseScreen == null) {

        this.pauseScreen = new PauseScreen();
        this.layerTop.addChild(this.pauseScreen);
        this.pauseScreen.setDelegate(this);
    }
}
```

E faça a chamada a ele na classe `GameButtons`:

```
if (sender.equals(this.pauseButton)) {
    this.delegate.pauseGameAndShowLayer();
}
```

Agora que já temos os métodos de pause criados na `GameScreen`, vamos voltar na classe `PauseScreen` e referenciar as ações dos botões, adicionando:

```
@Override
public void buttonClicked(Button sender) {
    // Verifica se o botão foi pressionado
    if (sender == this.resumeButton) {
        this.delegate.resumeGame();
        this.removeFromParentAndCleanup(true);
    }

    // Verifica se o botao foi pressionado
    if (sender == this.quitButton) {
        this.delegate.quitGame();
    }
}
```

Ao rodar o projeto, a tela de pause deve aparecer mas ainda temos trabalho a fazer.

11.6 Pausando os objetos

Temos toda a arquitetura preparada para o pause, mas algo muito importante ainda não foi feito. Ao rodar o projeto e apertar o pause, temos a tela aparecendo, porém, não temos os objetos parando.

Precisamos usar a classe `Runner` que controla o estado do game para isso. Será através das variáveis dessa classe que poderemos definir se os objetos como meteoros, tiros e avião devem se mover no `game loop`.

A lógica para isso será sempre igual. Apenas movimentaremos um objeto na tela de jogo se o jogo estiver rodando e não em pause. Além disso, só podemos criar novos elementos se essa combinação também for satisfatória.

Na classe `MeteorsEngine` adicione a verificação abaixo:

```
public void meteorsEngine(float dt) {
        if (Runner.check().isGamePlaying() &&
            ! Runner.check().isGamePaused()) {
            if (new Random().nextInt(30) == 0) {
                this.getDelegate().createMeteor(
                        new Meteor(Assets.METEOR));
            }
        }
    }
```

Na classe `Meteor` adicione a verificação abaixo durante o `update`:

```
public void update(float dt) {

        // pause
        if (Runner.check().isGamePlaying() &&
            ! Runner.check().isGamePaused()) {
            y -= 1;
            this.setPosition(screenResolution(CGPoint.ccp(x, y)));
        }
    }
```

Na classe `Shoot` adicione a verificação abaixo:

```
public void update(float dt) {
        if (Runner.check().isGamePlaying() &&
            ! Runner.check().isGamePaused()) {
            positionY += 2;
            this.setPosition(screenResolution(
                            CGPoint.ccp(positionX, positionY)));
        }
    }
```

Na classe `Player` adicione a verificação abaixo:

```
public void shoot() {
        if (Runner.check().isGamePlaying() &&
            ! Runner.check().isGamePaused()) {
            delegate.createShoot(new Shoot(positionX, positionY));
        }
    }

    public void moveLeft() {
        if (Runner.check().isGamePlaying() &&
            ! Runner.check().isGamePaused()) {
            if (positionX > 30) {
                positionX -= 10;
            }
            setPosition(positionX, positionY);
        }
    }
```

```
public void moveRight() {
    if (Runner.check().isGamePlaying() &&
        ! Runner.check().isGamePaused()) {
        if (positionX < screenWidth() - 30) {
            positionX += 10;
        }
        setPosition(positionX, positionY);
    }
}
```

Muito trabalho, não? A tela de pause pode enganar, mas como foi visto, é um ponto chave no desenvolvimento dos games. Rode o projeto e verifique o comportamento dessa nova tela no jogo.

11.7 Conclusão

Fazer uma tela de pause é normalmente uma parte que os desenvolvedores deixam de lado na construção de um game e depois percebem a complexidade de sua implementação. Essa tela usa conceitos que se propagam por todo o game e deve ser feita com cuidado e planejamento.

A notícia boa é que se você chegou até aqui você já tem os conceitos principais para desenvolver um jogo! Imagens, loop, camadas, sons, colisões, etc. Não são coisas fáceis, mas com a prática viram conceitos que se repetem por todo o game.

Figura 11.2: Tela de pause.

Capítulo 12

Continuando nosso jogo

Depois de saber como criar a base de um jogo e sua estrutura, montar os cenário e interações e criar telas de fluxo do game, muita coisa pode ser desenvolvida de acordo com a sua imaginação. A parte mais legal de agora em diante será pensar em ideias que possam tornar o jogo cada vez mais divertido e motivante. O mundo dos games é sempre um mundo de novidades, de inovações, em que cada nova ideia dá margem para milhares de novos jogos.

Esse capítulo mostrará sugestões de como tornar o game mais social, dinâmico e, quem sabe, rentável.

12.1 Utilizando ferramentas sociais

Para tornar o jogo mais engajador podemos adicionar diversas funcionalidades como rankings, onde os usuários disputam quem é o melhor, *badges* que servem como medalhas para provar conquistas durante o jogo e até monetizar o aplicativo com itens especiais.

Existem hoje diversas plataformas sociais que simplificam muito todo esse desenvolvimento, funcionando praticamente de forma *plug and play*.

Uma boa alternativa para nosso jogo é conhecer o `Swarm`. O `Swarm` é uma plataforma social utilizada por diversos games e traz as funcionalidade acima de uma forma simples de implementar.

Confira o site do `Swarm` em: http://swarmconnect.com/

Todo o *setup* da sua conta pode ser feito iniciando por esse link:
http://swarmconnect.com/admin/docs/setup

Figura 12.1: Plataforma Swarm Connect.

Os jogos que utilizam a plataforma `Swarm` têm na sua primeira tela uma tela de conexão, na qual com um simples aceite em um botão, um ID é gerado para aquele jogador. Fica a critério do jogador completar depois seus dados pessoais, como nome e apelido caso tenha interesse.

12.2 Highscore

Os *rankings* ou *highscores* no `Swarm` são chamados de *Leaderboards*. A ideia é criar um mural onde todos os usuários são ordenados para saber quem são os melhores.

A implementação é simples. A cada fim de jogo, fazemos uma requisição a *API* passando *ID* e pontuação.

O código fica parecido com:

```
SwarmLeaderboard.submitScore(LEADERBOARD_ID, meteorosDestruidos);
```

Você pode conferir a área sobre *Leaderboards* nesse link:
http://swarmconnect.com/admin/docs/leaderboard

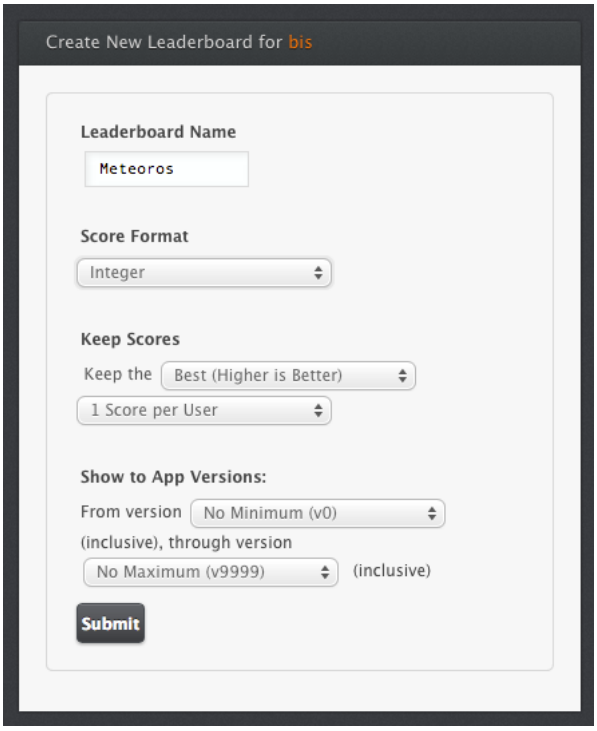


Figura 12.2: Leaderboards

12.3 Badges

Imagine se após destruir 10 meteoros o jogados ganhasse o título de `Piloto pró`, e após destruir 50 ganhasse o título de `Piloto Master`. Que tal, ao fim do jogo, entregar a *badge* `Santos Dumont` ao jogador? Essas criações tornam o jogo mais atrativo, com mais objetivos, e inclusive, podem ser compartilhadas em redes sociais.

A criação de *badges* usando o `Swarm` é também fácil. Você apenas define as *badges* do jogo e informa através da API quando isso ocorreu:

```
if (meteorosDestruidos == 50) {
    SwarmAchievement.unlock(PILOTO_MASTER_ID);
}
```

Esse `ID` será um número criado no site do Swarm.

Crie ideias interessantes de *badges* e implemente-as com `Swarm`. Para saber mais acesse a documentação na área *Adding Achievements to your App* pelo link:

http://swarmconnect.com/admin/docs/achievements

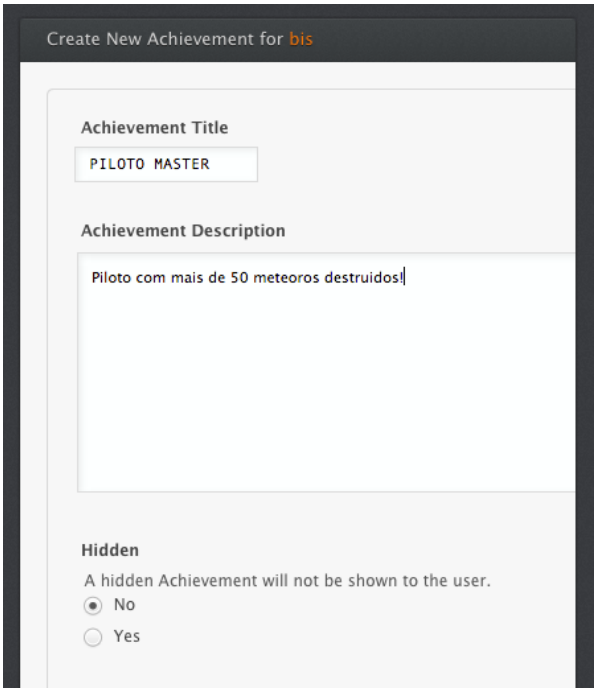


Figura 12.3: Criando badges.

12.4 Desafios para você melhorar o jogo

Aqui vão algumas sugestões que visam desafiar o leitor a solidificar os conhecimentos adquiridos com o livro e ampliar o engajamento do jogo 14 bis.

Novos tiros

Todo jogador adora incrementar sua munição. Crie um tiro duplo para o 14 bis! Esses tiros devem sair não mais em linha reta mas sim formando um angulo de 45 graus para cada lado. Interessante fazer esse tiro ser dado ao jogador após alguma conquista do game, como matar um número de meteoros, ou atirar em um elemento especial que deu esse poder!

Diferentes meteoros

Quem disse que todos os meteoros são iguais? Que tal implementar tamanhos diferentes de meteoros, que pontuam diferente de acordo com seu tamanho? Comece simples, meteoros maiores valem 2 pontos e meteoros menores continuam valendo 1. Com isso implementado, mude a imagem dos meteoros e até coloque esporadicamente outros elementos que podem valer 5 ou 10 pontos, mas que aparecem com um frequência menor. Vale também fazer com que os meteoros tenham velocidades diferentes entre si!

Armaduras

Morrer com apenas um meteoro é chato, mas podemos capturar um elemento que fortifica o avião! Crie um elemento que funciona como uma armadura e permite a colisão entre um meteoro e o avião. Lembre-se de mostrar ao jogador que ele está equipado com esse elemento, mudando a cor do avião por exemplo.

Efeitos nos sprites

Atualmente os objetos são estáticos no nosso game. Que tal adicionar efeitos como fazer o meteoro descer girando ou luzes no avião? Esses pequenos detalhes fazem o jogo parecer muito mais atraente.

12.5 Como ganhar dinheiro?

O mundo dos games move um valor enorme e é hoje uma das maiores movimentações financeiras do mundo. Esse livro não visa trazer análises sobre esse mercado financeiro, porém hoje as cifras dos games superam os números do cinema.

Para monetizar o game você pode seguir por diversas abordagens, como estabelecer um valor de venda quando o usuário baixar. Porém, no mundo dos smartphones essa estratégia não é muito utilizada.

Uma forma bem interessante de ganhar dinheiro com o jogo é deixando-o gratuito ou cobrando um valor bem baixo, com o qual o usuário vai instalar o jogo sem muito esforço. Após isso, o jogo deve cativar o usuário e então pode oferecer itens a serem comprados que melhoram a experiência e performance do jogo.

O `Swarm` oferece uma forma bem interessante de monetizar o game e fácil de aplicar. Tendo toda a documentação e API pronta, você pode adicionar esses itens com poucas linhas de código, como nas linhas abaixo, que indicam a venda de um

item:

```
SwarmStoreListing.purchase(context, SUPER_BOMBA_ID);
```

Para saber mais sobre monetização do jogo utilizando o `Swarm` acesse a área *Adding a Swarm Store to your App* no seguinte link:

http://swarmconnect.com/admin/docs/store

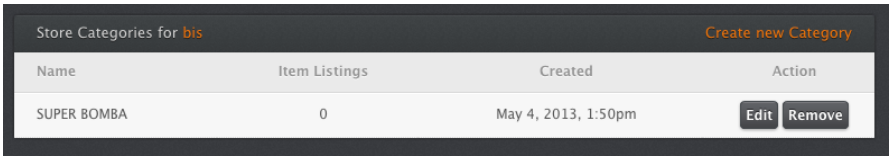


Figura 12.4: Exemplo de itens pagos.

12.6 Conclusão

Desenvolver jogos é uma tarefa complexa e ao mesmo tempo divertida, além de ser uma ótima maneira de elevar o conhecimento de programação. Atualmente, os celulares modernos trouxeram uma oportunidade única para que desenvolvedores possam ter essa experiência, criando seus próprios games, uma revolução similar a que a web e os computadores pessoais geraram alguns anos atrás.

Além disso, criar jogos é um exercício de criatividade que permite explorar a nossa imaginação e criar histórias interativas e únicas. Esse livro tenta compartilhar esses conhecimentos e ideias, esperando ser de fato útil a todos aqueles que estão ingressando nesse mágico mundo de jogos para Android.

Fica novamente o convite para você participar da nossa lista de discussão:

https://groups.google.com/group/desenvolvimento-de-jogos-para-android

E você pode tirar dúvidas pelo fórum do GUJ.com.br:

http://www.guj.com.br/

Boa diversão!